Linux Journal Issue #55/November 1998

Features

News & Articles

XML has been attracting a lot of attention recently. This article provides a five-minute overview of XML and explains why it matters to you.

More Flexible Formatting with SGMLtools *by Cees de Groot*

A brief overview of the latest SGMLtools is presented by one of its developers.

Tcl/Tk: The Swiss Army Knife of Web Applications *by Bill Schongar*

Tcl/Tk offers many uses to the web programmer. Mr. Schongar describes a few of them.

Reviews

QuickStart: Replication & Recovery v1.2 An overview and review of this replication and recovery product. *by Daniel Lazenby*
Informix on Linux: First Impressions Notes on installing and configuring Informix's port to Linux. *by Fred Butzen*
Structuring XML Documents *by Terry Dawson*
Linux Kernel Internals, Second Edition *by Karl Majer*

Columns

Linux Apprentice Beginner's Guide to JDK *by Gordon Chamberlin*

Beginner's Guide to JDK This article covers the use of the Java Development Kit on a Linux platform. It includes a general introduction to Java, installing the JDK 1.1.6, compiling Java support into the Linux kernel, writing a simple Java program and studying an example.

Take Command init *by Alessandro Rubini*

init init is the driving force that keeps our Linux box alive, and it is the one that can put it to death. This article is meant to summarize why init is so powerful and how you can instruct it to behave differently from its default behaviour. (Yes, init is powerful, but the superuser rules over init.)

Linux Means Business Linux for Internet Business Applications *by Uche Ogbuji*

Linux for Internet Business Applications A look at how one company is moving ahead by using Linux to provide Internet services to its clients.

System Administration High Availability Linux Web Servers *by Aaron Gowatch*

High Availability Linux Web Servers If a web server goes down, here's one way to save time and minimize traffic loss by configuring multiple hosts to serve the same IP address.

Linux Gazette The Roxen Challenger HTTP Web Server *by Michel Pelletier*

The Roxen Challenger HTTP Web Server A review of the easy-to-install web server written in Pike.

Departments

Letters to the Editor
From the Editor *by Eric S. Raymond*

Open Source's First Six Months

LINUX JOURNAL

FastCGI: Persistent Applications for Your Web Server

Paul Heinlein

Issue #55, November 1998

FastCGI allows Apache to run and manage persistent CGI-like scripts, overcoming CGI's worst shortcomings.

The Common Gateway Interface is nearly as old as the web server itself. NCSA added the CGI 1.0 specification to version 1.0 of the granddaddy of all web servers, **httpd**. CGI 1.1, the current specification, was added to the 1.2 release. Every popular web server package developed since then has incorporated CGI as a way—usually *the* way—for web-borne visitors to access server-based executables.

CGI vs. FastCGI

CGI works well with small or infrequently used programs in which their sole function is to respond to one-time requests, such as processing simple information from HTML forms. There's no sense in clogging up your memory or process table with small applications invoked only a few times an hour.

The opposite is true, however, for complex or frequently used programs. Your web server can slow to a crawl if your site depends on a script with a long initialization process; in particular, one that involves connecting to a database or reading and structuring information from large text files. Speed issues are even more critical for small sites with servers that also process mail, FTP or DNS requests.

CGI applications must be launched anew with each invocation, a limitation that leads to two problems. First, the hardware and operating system have to deal with the overhead of creating a process for every CGI request. Second, CGI scripts can't handle persistent variables or data structures; they must be rebuilt with each invocation.

Open Market's FastCGI interface is one way to overcome CGI's limitations. FastCGI applications are invoked via URLs just like their CGI counterparts. The difference is that they're persistent; they function like servers within a server. FastCGI offers benefits in three areas:

- **Speed**: a FastCGI script goes through only one process-creation cycle. Initialization of data and database connections are done only once. A further benefit of FastCGI is that it can connect to processes running on remote machines, taking the processing burden off the main server.
- **Persistence**: even if you don't have access to an SQL server, FastCGI enables many database-like functions by storing your complex methods, objects and variables in RAM. Data can be stored across sessions, providing a workaround for the statelessness of HTTP connections.
- **Process management**: Apache's implementation of FastCGI gives the server daemon the ability to take care of FastCGI applications, automatically restarting them should they die off. Other servers may share this ability; my experience is limited to Apache.

Other CGI Alternatives

FastCGI is not the first—nor, I'm sure, will it be the last—approach to move beyond CGI. Most web servers have APIs that allow developers to write new functions right into the server. Doug MacEachern's **mod_perl** module allows the Perl runtime library to be compiled directly into Apache, giving hackers the ability to write server modules entirely in Perl.

I prefer FastCGI to its alternatives for four reasons:

1. It's not language-specific. **mod_perl** and proprietary APIs all dictate that the developer use a certain programming language. FastCGI applications can currently be written in Perl, C/C++, Java or Python, and the standard is flexible enough that other languages could be added in the future.
2. It's not server-specific. Actually, the *implementations* of FastCGI are server-specific, but the standard is not tied to one software package. FastCGI is currently supported on Apache, Roxen, Stronghold, and Zeus; a commercial variation is available for Netscape and Microsoft servers from Fast Engines, http://www.fastserv.com/.
3. FastCGI applications don't run in the server's name space. If a FastCGI application dies, it doesn't take the server down with it. Also, since FastCGI scripts run as separate processes, they don't increase the size of the server executable.
4. It's scalable. FastCGI scripts can be configured to run remotely via a TCP/IP connection, providing a method for load sharing.

Software Requirements

My server platform is a fairly standard Linux box: a 120MHz Pentium with 64MB RAM running the 2.0.27 kernel. If you already get decent performance from your hardware, you won't have any trouble with FastCGI.

As to software, I use Apache and Perl; the material below is unabashedly biased in their favor. If you want to do your coding in C/C++, tcl, Java or Python, or if you want to use different server software, I suggest you visit http://fastcgi.idle.com/ for further information. On the other hand, most of the coding hints I'll be providing are applicable to any language.

To use Perl and Apache, you'll need to do some recompiling. Apache needs to be rebuilt with the FastCGI module. You'll also have to compile a Perl module. A few months ago you would have needed to rebuild Perl—I'll provide instructions in case you need or want to do so—but now can probably get by on your current Perl build. Even if you're not an accomplished C programmer, however, the compilation process is fairly painless. Here's what you'll need:

- A C compiler of recent vintage: I've used **gcc** 2.7.2.1 without any problems.
- The Apache source code: I use Apache 1.3.0 (1.3.1 is the current revision). Apache comes with most Linux distributions, or you can download it from http://www.apache.org/.
- The Perl 5.004 or 5.005 source code: I strongly recommend that you upgrade if you're using an older version. If nothing else, it's a good opportunity to take a peek at the new and improved Perl home page at http://www.perl.com/.
- The **mod_fastcgi** source code: The current version (as of September 3, 1998) is 2.0.17. It's compatible with Apache 1.3.1 and is available at http://fastcgi.idle.com/fastcgi.tar.gz.
- Documentation and sample scripts: These are available with the The FastCGI Developer's Kit, links to which are provided at http://fastcgi.idle.com/.
- Sven Verdoolaege's FCGI.pm (http://www.perl.com/CPAN/modules/by-module/FCGI/) handles the Perl-to-FastCGI interaction; this is the module on which my examples are based. Alternatively, you can use Leonard Stein's CGI::Fast module included in the standard Perl distribution (in which case you'll need to tweak my example code a bit).
- You may need AT&T's freely distributed Safe/Fast I/O (**sfio**) libraries, available from http://www.research.att.com/sw/tools/sfio/. Until last June, Perl needed to be rebuilt with sfio to be able to handle FastCGI I/O streams. The new versions of FCGI.pm work without sfio (at least I've had no trouble), but some posts to the fastcgi-developers mailing list suggest

that there may still be a few kinks in the new module. My recommendation is that you try FastCGI on a stock Perl build before resorting to building it with sfio.

Once you've gathered the necessary source code, you'll be ready to spend some quality **make** time. Compilation is done in two segments: Perl and Apache. Either can be done first, but within each segment the steps have to be completed in a certain order.

Compiling Perl with sfio

Unless you know you need to recompile the Perl binary, you can skip down to the "Compiling FCGI.pm" section. If, however, you do need to recompile Perl, it's helpful to know a few things.

To begin the Perl compilation process, unpack and build sfio; the README will tell how. You'll also have to update your **$PATH** to include one of the newly created subdirectories; this is somewhat unusual, but required.

Second, build, test and install Perl. It can take awhile to work through Larry Wall's Configure script, but there are a few items for which you should not choose the default answer:

1. To the question "Directories to use for library searches", answer **$* *$sfio*/lib** (where $sfio is the directory in which you unpacked **sfio**). The default answer to the next question, "Any additional libraries?", should now include **-lsfio**.
2. To the question "Any additional cc flags?", answer **$* -I$sfio/include**.
3. To the question "Any additional **ld** flags (*not* including libraries)?", answer **$* -L$sfio/lib**.
4. To the question "Use the experimental PerlIO abstraction layer?" answer yes.
5. To the question "perl5 can use the sfio library, but it is experimental. You seem to have sfio available, do you want to try using it?", answer yes.

I've never had trouble re-compiling Perl in this way, but the fastcgi-developers mailing list archive (available from http://www.findmail.com/list/fastcgi-developers/) has plenty of messages from people who have. A fairly complete set of directions for recompiling Perl with **sfio** can be found at http://fastcgi.idle.com/fcgi2.0b2.1/doc/fcgi-perl.htm.

Compiling FCGI.pm

Regardless of whether you had to recompile Perl, you'll need to unpack, build, test and install FCGI.pm according to the instructions provided with the source

code. You can be fairly sure you're on the right track if FCGI.pm passes **make test**.

Compiling Apache

Before dealing with the Apache distribution, unpack the **mod_fastcgi** source code. Read the INSTALL file, which details the two ways to configure, compile, and install Apache. The first, the Apache Autoconf-style Interface (APACI), is new to version 1.3. The second is the tried-and-true manual configuration we all know and love. Then unpack Apache's source code and configure it for the FastCGI interface:

1. Copy or move the mod_fastcgi distribution directory to <apache_dir>/src/modules/fastcgi.
2. Configure Apache using either APACI or the manual method as detailed in the mod_fastcgi INSTALL file. I'm pretty accustomed to dealing with the Configuration file, so I usually do it the old way.
3. If you're using Apache 1.2.x or 1.3.x and you're not running out of RAM, try uncommenting the line containing **mod_rewrite**. This is a tremendous extension to Apache that allows it to parse incoming URLs as regular expressions. See the sidebar, "Health and Beauty the Rewrite Way", for my line of reasoning in this regard.
4. Run **make**.

Configuring Apache

Apache gets all of its runtime directives from three files found in $apache/conf: access.conf, httpd.conf and srm.conf. Following the suggestion given by Ben and Peter Laurie in *Apache: the Definitive Guide*, I put all directives in httpd.conf and don't use the other two files. If you use all three files, the configuration changes will occur in srm.conf.

Before doing any configuration, you'll need to read the documentation included with the mod_fastcgi source code. The docs/mod_fastcgi.html document is somewhat dated, but still very useful for getting you started. No author is listed, but I'd gladly buy him or her a beer for putting together a truly excellent resource, and thereby making my job much simpler.

Let me also say that you should have more than a passing familiarity with httpd.conf before altering it. Take a good look at the documentation that comes with the source code or buy yourself a copy of the Lauries' book.

The FastCGI configuration directives (see sidebar "Configuring Apache for FastCGI") allow you to accomplish two essential tasks.

First, define the local pathway to the FastCGI applications using the **AppClass** directive and/or the remote connection host and port number via **ExternalAppClass**. **AppClass** is responsible for starting and managing processes that run locally.

Second, associate your FastCGI applications with the proper handler or MIME type so that dealings with these files are handled by mod_fastcgi. Associate the handler “fastcgi-script” with a file or files based on location (**SetHandler**) or file extension (**AddHandler**). Alternately, you can associate the MIME type

```
application/x-httpd-fcgi
```

with a file or files based on location (**ForceType**) or file extension (**AddType**).

Note that your FastCGI applications cannot go into the normal CGI directory specified by **ScriptAlias**. Apache's way of assigning priorities leads it to attempt to handle any and all files in the CGI directory with the standard CGI module, which won't work with FastCGI applications.

Writing Scripts

In many ways, writing FastCGI scripts is not very different from traditional CGI programming. You must specify a **Content-type** (typically, “text/html”) if you're providing content. You can use **Location** and **Status** to specify redirects or other HTTP messages. Also, you have normal access to the **%ENV** hash.

From within scripts, STDIN and STDOUT can be accessed, but only in standard ways. The FastCGI library manipulates those data streams quite heavily; you can **print** without trouble, but more advanced operations will fail. You can't, for example, send a reference to a typeglob (a symbol table entry) of STDOUT (***STDOUT**) to a forked process. In fact, FastCGI is fairly scornful of forking, and I haven't heard any reports at all from someone trying to run it on a thread-enabled version of Perl 5.005.

The main difference, structurally speaking, between CGI and FastCGI scripts is that the main body of code is placed within a **while** loop, one which hopefully never ends. The basic structure of a FastCGI script is pretty much the same regardless of its task:

1. Initialize variables and connections to databases, daemons, etc.
2. Do the loop.
3. Provide for cleanup so you can exit gracefully when needed.

Although FastCGI will force few substantive changes in your code, it will likely change your perspective on what makes a good script. Some of the lessons I've learned while developing FastCGI applications are:

- **Think clean**. Typical CGI scripts don't need to be excessively concerned with memory leaks or sloppy variable scoping. FastCGI scripts, since they're persistent, have to keep a tighter rein on things.
- **Think big**. We're used to thinking of CGI scripts as fast one-timers that should define the fewest functions necessary to get the job done. With FastCGI, it's usually better to have lots of functionality in one script; you have easier access to shared data and fewer PIDs littering your process table. I try to use the main script (the one specified in httpd.conf) as a distribution center, jobbing out all the real work to modules. Doing so makes it easy to extend the main script's functionality with just an extra line or two of code; all your tweaking can be done on the module.
- **Think long-term**. You want your process to keep running, so it's wise to not let your script **die()** or **croak()**. Catch the return value of any statement whose failure might prove fatal (such as **open()**) and rely on error messages and flow control to keep the loop running.

Ad Rotation Made Easy

Webmasters of commercial sites hate to admit it, but getting advertisements on-line is an increasingly unavoidable fact of the job. If you have multiple sponsors in a rotation, or if your sponsors each have multiple ads, there's no way to hardcode the ad into a page stored on disk. Of course, this is true for any information likely to be presented on a rotating basis: news, current specials or random links.

The rotate.fcg script shown in Listing 1 provides a bare-bones approach to meeting that need. It provides a persistent array of ad information that can be inserted wherever you choose on any disk-based document. It also allows the ad array to be updated without having to re-start the script (although this technique won't work if you're running multiple instances of the script).

Based on the Apache configuration shown in the Apache sidebar, the URL to invoke the script is http://www.yoursite.com/fastcgi-bin/rotate.fcg?*page*.html, where "*page*.html" is the name of a document into which you'd like to insert an ad. *page*.html can contain one or more instances of an HTML comment that serves as a placeholder for the ad:

```
<!-- Ad Here -->
```

Using an HTML comment in this capacity means that the document will display correctly, even if you have no ad to put there yet.

The script's opening section scopes and initializes all variables to be used for the life of the process. Three things are worthy of note in this section. First, since we initialize **@ads** outside the loop, it will stay persistent for the life of the script. Second, we need to initialize the **%ENV** array ourselves, lest we find it empty later on down the line. Third, we set **$|** to a non-zero number, because we want to flush STDOUT every time the script is invoked.

Right before the script enters the main loop, it initializes the array of ads by calling the **initialize** routine. This routine reads a text file of the sort shown in Listing 2. The data for each sponsor are temporarily put into the **%sponsor** hash, formatted into HTML and **push**ed into the **@ads** array. If the text file can't be opened, the routine returns an empty array, allowing the script to run anyway.

The main action takes place in the loop labeled **REQUEST**. The **while** command is the only place the script interacts explicitly with FCGI.pm. It's also the only substantive difference between a FastCGI script and a traditional one. Regardless of the language you use for FastCGI programming, a loop like this one will be the structure in which you frame the script's main process.

Once in the loop, the first task is to allow the webmaster to re-initialize the ad array on the fly. In the example script, this is accomplished by placing a request to http://www.yoursite.com/fastcgi-bin/rotate.pl?reload. To provide a little security, the script allows re-initialization only from the web server. If you're running multiple instances of a script, you'll have to accomplish this by some other means: restarting Apache with **kill -USR1**, reloading the data file if its timestamp has changed, etc.

If you used a script like this to run current news headlines, it would be easy to post new updates to your site several times each day by adding them to the text file and re-initializing the array.

The loop's second task is to make sure that the requested file can be opened. If it can't, the script calls a routine (not included in my example) that would send off a “File Not Found” message. By providing its own error message, the script can recover gracefully from a bad request without having to die off. If it is available, the requested document is assigned to the **@doc** array.

Next, an ad is pulled off the front of the **@ads** array, assigned to **$ad**, then pushed to the back of the array. The script retains a copy of the ad, even though it's been put back in the array.

Fourth, the script cycles through the document looking for any instances of **<!-- Ad Here -->**. When it finds one, it substitutes the **$ad** for it. If the text file containing the ads is empty or unopenable, or if the requested page has no place for an ad, no substitutions are made.

Finally, the script prints the appropriate HTTP header, sends off the document and heads back to the front of the loop to wait for the next request.

Finis

My example script doesn't tackle many of the tasks at which FastCGI excels: persistent database connections, format translation (e.g., SGML to HTML) or providing common HTML page headers and footers. At the site I manage, I use FastCGI to do all these things and more.

I've found that a FastCGI application can perform its duties, including multiple SQL queries, and deliver a page on the fly only slightly slower than the server can deliver static documents. On a 10Mbps LAN connection the speed difference is perceptible, but just barely, and only if I'm looking for it. Over a 128Kbps or slower connection, I notice no difference.

I still use CGI to perform simple, infrequently needed tasks. A CGI script doesn't hog system resources for very long. For complex, frequently invoked tasks, FastCGI provides a great combination of flexibility and speed.

The two listings referred to in this article are available by anonymous download in the file ftp.linuxjournal.com/pub/lj/listings/issue55/2607.tgz.

Paul Heinlein (heinlein@teleport.com) lives with his family near Portland, Oregon and is Webmaster at http://www.computerbits.com/. When he and his daughter aren't playing CD-ROM-based games, Paul indulges his odd hankering for Lutheran theology and hymnody.

Archive Index Issue Table of Contents

Advanced search

An Introduction to JDBC

Manu Konchady

Issue #55, November 1998

Mr. Konchady presents some of the benefits of using Java over CGI as well as the basics of managing a departmental database with Java.

In 1996, Sun released a version of the Java Database Connectivity (JDBC) kit. This package allowed programmers to use Java to connect, query and update a database using the Structured Query Language (SQL). The use of Java with JDBC has advantages over other database programming environments. Programs developed with Java and JDBC are platform and vendor independent, i.e., the same Java program can run on a PC, a workstation, or a network computer. Also, the database can be transferred from one database server to another and the same Java programs can be used without alteration. This article discusses how JDBC can be used with the MySQL database. (See “At the Forge” by Reuven Lerner in the September, October and November 1997 issues of *LJ*.)

A number of technologies have been developed for databases, such as transaction processing, triggers and indexes, which are supported by JDBC. However, these topics are beyond the scope of this article. Since the JDBC package is relatively new, the tools that database developers expect, such as report generators, query builders and form designers are available. In the near future, more tools should be available. Despite the lack of tools, it is possible to develop highly interactive web pages using the JDBC API without much complexity.

Design

The idea behind JDBC is similar to Microsoft's Open Database Connectivity (ODBC). Both ODBC and JDBC are based on the X/Open standard for database connectivity. Programs written using the JDBC API communicate with a JDBC driver manager, which uses the current driver loaded.

Two architectures can be used to communicate with the database (see Figure 1). In the first one, the JDBC driver communicates directly with the database. The driver connects to the database and SQL statements on behalf of the Java program. Results are sent back from the driver to the driver manager and finally to the application.

In the other, the JDBC driver communicates with an ODBC driver via a “bridge”. A single JDBC driver can communicate with multiple ODBC drivers. Each of the ODBC drivers execute SQL statements for specific databases. The results are sent back up the chain as before.

Figure 1. Database Communication Architecture

The JDBC/ODBC bridge was developed to take advantage of the large number of ODBC-enabled data sources. The bridge converts JDBC calls to ODBC calls and passes them to the appropriate driver for the backend database. The advantage of this scheme is that applications can access data from multiple vendors. However, the performance of a JDBC/ODBC bridge is slower than a JDBC driver alone would be, due to the added overhead. A database call must be translated from JDBC to ODBC to a native API.

Figure 2. Accessing a Database from an Applet

Fewer operations are required to use a JDBC driver without a bridge. In Figure 2, the steps to access a database using a JDBC driver from an applet are shown. The Gwe JDBC driver is used with the MySQL database. The JDBC driver classes are first downloaded from the Gwe host site. Next, the applet logic passes a JDBC call to a driver manager which in turns passes the call to a JDBC driver. The JDBC driver opens a TCP/IP connection with the MySQL database server. Data is transferred back and forth via the connection. When database processing is complete, the connection is closed.

The JDBC API can be used in applets and stand-alone applications. In addition to the usual restrictions for applets, only connections from the same server from which the applet was downloaded are accepted. If the web server and database server are not on the same machine, the web server must run a proxy service to route the database traffic. Stand-alone applications can give access to local information and remote servers.

Installation

Installation of the JDBC driver for the MySQL database is simple. The software can be downloaded from Gwe Technologies at http://www.gwe.co.uk/. The copyright statement allows the redistribution of source and binary, but is not

identical to the GNU license. Download the file, exgweMysqlJDBC.0.9.2-src.tar.gz. It contains the source and class files for the Gwe MySQL JDBC driver.

The use of JDBC requires access to java.sql classes which were not available in the pre-1.1 Java Development Kit. These classes have been renamed and included in the /exjava directory. Another directory, /exgwe, contains the source and classes for the Gwe MySQL JDBC driver. These classes make use of the classes in the exjava directory. Add the directory in which the tar file was unpacked to the CLASSPATH environment variable, and installation of the JDBC driver is complete.

A Sample JDBC Program

This example, shown in Listing 1, assumes you have some familiarity with Java. The Java code loads the JDBC driver class, establishes a connection with a database, builds an SQL statement, submits the statement and retrieves the results. A database and a table populated with some data must exist.

In the first executable line, the class object for the JDBC driver is loaded by passing its fully qualified name to the **Class.forName** method. This method loads the class, if it is not already loaded, and returns a Class object for it. In the next line, the database URL string is constructed in the form **jdbc:*subprotocol_name*: hostname:*port/database_name/other parameters***. The subprotocol name is **mysql**, since we are using the MySQL database. The host name is **localhost** in this example, but can also be an Internet host name or IP address. The port number for the MySQL server is **3306** and the name of the database is **test**. Other parameters can be passed in the database URL, such as user ID and password.

A connection object is obtained via a call to the **getConnection** method of the driver manager, allowing use of the JDBC driver to manage queries. The user ID and password are in clear text in the file. The password is encrypted by the JDBC driver before passing the information to the MySQL server. A statement object is required to issue a query. The statement object is obtained by calling the **createStatement** method of the connection object.

The SQL query is stored in a string and passed to the **executeQuery** method of the statement object, which returns a **ResultSet** object containing the results of the query. The **next** method of the **resultSet** object moves the current row forward by one. It returns false after the last row. This method must be called to advance to the first row, and can be called in a loop to retrieve data from all matching rows. The **resultSet** object contains a number of methods to extract data from a row. For example, to retrieve a string, the **getString** method is used. Similarly, to retrieve an integer, the **getInt** method is used. Other methods to

retrieve a byte, short, long, float, double boolean, date, time and a blob are included. The **getBytes** method can be used to retrieve a binary large object (blob). The parameter to these methods is either an integer or a string. The integer is the column number of the row retrieved. Not all columns of a table need to be retrieved. The string is the name of the column label.

Once data has been extracted from the **resultSet** object, it is closed. Another SQL query can be issued and the **resultSet** object can be reused. The statement object can also be reused. The statement and connection objects are closed when database retrieval is complete. This simple example illustrates the process of retrieving data from a database table. It is also possible to update tables and obtain information about tables. When updating tables, the **executeUpdate** method of the statement object is used. For example:

```
String query = "update test_table set phone =
        999-9999 ";
query += "where name = \"John Smith\"";
stmt.executeUpdate( query );
```

Database Metadata

JDBC can be used to obtain information about the structure of a database and its tables. For example, you can get a list of tables in a particular database and the column names for any table. This information is useful when programming for any database. The structure of a database may not be known to the programmer, but it can be obtained by using metadata statements—SQL statements used to describe the database and its parts.

Two types of metadata can be retrieved with JDBC. The first type describes the database and the second type describes a result set. The **DatabaseMetaData** class contains over a hundred methods to inquire about the database, some of which are quite exotic. A common method is the **getTables** method.

```
DatabaseMetaData dmd = con.getMetaData();
ResultSet rs = dmd.getTables( null, null, null,
        new String[] {"TABLE"} );
```

The parameters passed to getTables are, in order, a catalog (group of related schemas), a schema (group of related tables) pattern, a table name pattern and a type array. Some of the types include table, view and system table. If **null** is passed, no pattern is used to limit the metadata information retrieved. Some of the other methods include **getDataProductVersion**, **getTablePrivileges** and **getDriverName**. The result set **rs** contains information about all the tables in the database. Each row contains information about a table. For example, the third column of any row of the result set is the table name string.

Useful metadata can be obtained about a result set after the execution of a query. When a result set is obtained after the execution of a query, the

metadata statements can be used to extract information such as the number of columns, column types and width.

```
ResultSet rs = stmt.executeQuery("Select * from test_table");
ResultSetMetaData rsmd = rs.getMetaData();
```

The **rsmd.getColumnCount()** method returns the number of columns in the test_table and the **rsmd.getColumnLabel(i)** method returns the name of the *i*th column. Similarly, the **rsmd.getColumnDisplaySize(i)** method returns the width of the *i*th column. A number of other methods described in the JDBC API can be used to extract all types of information about a table.

Example

At Mitre, we collect information from the Internet using commercial search engines such as Altavista and Lycos for a variety of topics. This information is stored as a collection of text documents for any topic and can be searched via keywords. We use the public-domain search engine Glimpse from the University of Arizona to index and search the document collection.

Some of the document collections can be fairly large (over 1500 documents). If a common keyword is entered, the list of matching documents will be large. We display the results from the search engine using Java and JDBC to avoid scanning long lists of matching documents. Java was used to build a 3-D space and plot circles at locations representing the frequency of the occurrences of keywords in a document. JDBC was used to retrieve the titles of the documents stored in a table. Passing all the titles of all documents in the collection as parameters to the Java applet would significantly increase the time to load the applet.

Figure 3. Documents Represented as Circles in 3-D Space

Glimpse returns the frequency of occurrence of a keyword in a document. We use that number to locate a circle representing the document in 3-D space (see Figure 3). Each axis represents a keyword. If fewer than three keywords are entered, documents will be displayed in a plane or on a line. If more than three keywords are entered, three or fewer keywords must be chosen in order to display matching documents.

The frequency of occurrence of keywords is normalized for each axis, and the frequencies of keywords in documents are passed as parameters to the applet. The color of the circle was computed based on the position of the circle in the three axes. Red is used for documents on the z-axis, green for documents on the y-axis and blue for documents on the x-axis. Brighter shades of the three

primary colors are used for documents with higher keyword frequencies. A mix of the primary colors is used for circles which contain more than one keyword.

JDBC is used to retrieve the titles for documents containing non-zero occurrences of the keywords. This number is usually fewer than the total number of documents when a fairly unique keyword is used. When the mouse is located over the document, a window is displayed with the document's title. Sometimes, more than one document can have the same frequency of occurrence of a keyword. In such cases, the window displays multiple titles of documents. The color of the circle changes to white to indicate the document where the mouse is located. An option to click on a box in the window is provided and will retrieve the text corresponding to the document in a separate window.

Conclusion

This article has described the basics of working with JDBC under Linux: the design of JDBC, the installation of JDBC for MySQL and example code to retrieve/store data. Metadata statements can be used to interrogate the structure of a database and its tables. Finally, we looked at an example using a search engine with JDBC and Java. Viewing the results from a Java applet made the user's task more interesting than it would have been through a CGI program.

The listing referred to in this article is available by anonymous download in the file ftp.linuxjournal.com/pub/lj/listings/issue55/2846.tgz.

Manu Konchady (manuk@mitre.org) works at Mitre Corporation developing software for information retrieval. As the lone user of Linux in a group of 50, he is striving to promote its many benefits.

Archive Index Issue Table of Contents

Advanced search

Perl Embedding

John Quillan

Issue #55, November 1998

An overview of what is needed to make your favorite application Perl-enabled and how to avoid some obstacles along the way.

This article describes my experience embedding Perl into an existing application. The application chosen, **sc**, is a public-domain, character-based spreadsheet that often comes as part of a Linux distribution. The reason for choosing sc was twofold. First, I use sc for any spreadsheet-type tasks I have. Second, I was somewhat familiar with the source, because I once added code in order to format dates the way I wanted. Besides, I always thought it would be nice if sc had some sort of macro language—everything should be programmable in some way.

Getting Started

The first thing I did was to get the sc 6.21 source and compile it on my machine. This ensured that everything worked from the start, before I started making modifications to the code.

The next thing was to add the necessary code to sc.c to embed the Perl interpreter. The basics of this were:

- Add the following include files

```
#include <EXTERN.h>
#include <perl.h>
```

- Add the variable for the Perl interpreter

```
static PerlInterpreter *MyPerl;
```

- Put the code shown in Listing 1 in **main** in the file sc.c.
- Update Makefiles to use the correct parameters. This consisted of adding CC options and linker options derived from the following commands:

```
perl -MExtUtils::Embed -e ccopts
perl -MExtUtils::Embed -e ldopts
```

These commands give you the compiler and linker options that your version of Perl was compiled with.

Nothing else needs to be done; the Perl interpreter is now in the code. Obviously you can't do anything yet, but you can work out any compilation problems. Right away, I had a few problems with some **#define** statements and a prototype for main. **EXT** and **IF** were the two offending **#define**s. I fixed these by appending "sc" to the end of them wherever they occurred in the original sc code, to make them unique. If you were writing an application from scratch, it would not be a bad idea to prepend a common prefix to each **#define**.

Perl, on the other hand, expected main to have a third argument, **env**, so I added it. I am still not sure where this argument comes from, but it doesn't seem to create any problems.

Running Perl Code

Once the base interpreter compiled successfully, I needed a way to call the functions. I looked at the sc source and found that one of the keystrokes, **ctrl-k**, was free for my use. I used this as my "call Perl" key-command macro, with macros from 0 to 9 defined. This combination calls predefined Perl subroutines called sc_macro_[0-9], when defined. The code in Listing 2 adds this functionality.

The function **call_sc_perl_macro** checks first to see if the subroutine exists with **perl_get_cv**. If null is not returned, it calls the function which has the name **sc_macro_#** where # is a digit from 0 to 9.

The **perl_call_va** function comes from Sriram Srinivasan's book *Advanced Perl Programming*, published by O'Reilly. This code was used to expedite my ultimate goal of embedding Perl into sc. The code for perl_call_va can be found in the file ezembed.c.

With sc compiled, I proceeded to test the interpreter by creating dummy macros in the file sc.pl to write some data to temporary files. Everything worked fine, which told me the Perl interpreter was working inside of sc.

Automating sc

With a working Perl interpreter embedded into sc and the ability to call Perl "macros", the interfaces to the C functions in sc needed to be created to do useful work. Fortunately, sc is laid out nicely enough that, for the most part, all one has to do is wrap an already existing function and interface with its internal command parser.

The first thing I thought might be useful is to move the current cell around. Without that ability, I would be able to operate only on a single cell, which is not very useful. Besides, it was one of the least complicated sections of code and provided a good start.

The code for **sc_forward_row** is shown in Listing 3 and found in sc_perl.c. Before I describe this code, let me give you a quick overview of how Perl treats scalars. Each scalar has many pieces of information, including a double numeric value, a string value and flags to indicate which parts are valid. For our purposes, the scalar can hold three types of values: an integer value (IV), a double value (NV) and a string value (PV). For any scalar value (SV), you can get to their respective values with the macros SvIV, SvNV and SvPV.

Now, in the Listing 3 code, XS is a Perl macro that defines the function. dXSARGS sets up some stuff for the rest of XSub, such as the variable **items** that contains the number of items passed to Xsub on the Perl argument stack. If the argument count does not equal 1, XS_RETURN_IV returns 1 to Perl to indicate an error. Otherwise, the top element of the Perl argument stack, ST(0), is converted to an integer value and passed to the **forwrow** function.

Note that all of the XSub code was generated by hand. Some of this work can be done with Perl's **xsubpp** or with a tool called **swig**, but in this case, I felt it was simpler to code it myself.

Finally, tell the Perl interpreter about this Xsub with the statement:

```
newXS("sc_forward_row",sc_forward_row,"sc_perl.c");
```

The first argument is the name of the subroutine in Perl. The next argument is the actual C routine (in this case they are the same, but they don't have to be). The last argument is the file in which the subroutine is defined, and is used for error messages. I chose to create all of the **newXS** functions by parsing my sc_perl.c file with a Perl script, so that I would not have to do two things every time I added a new XSub.

Dynamic Loading of Modules

The next thing I wanted was the ability to dynamically load other Perl extensions such as the Tk extension, database extensions or anything else that might prove useful. This requires an **xs_init** function in place of the **NULL** in perl_parse as shown below.

```
perl_parse(MyPerl, xs_init, 2, my_argv, env);
```

To create the xs_init, I used the following code:

```
perl -MExtUtils::Embed -e xsinit -- -o - >xs_init.c
```

The function of xs_init is to initialize the statically linked extension modules. The only module I statically linked is the DynaLoader module. With this module, we can dynamically load everything else. When I initially did this, I had numerous problems. They turned out to be linked to the version of Perl I was using (5.003_07). After I installed 5.004_04, everything worked fine.

Other Problems

One of the first problems I ran into was the fact that Perl redefined **yypars** to be **Perl_yypars**. I fixed this by putting new **#define** statements around places where I used sc's yypars. This created a lot of compiler warnings, but did allow the code to compile correctly.

The other problem I encountered was with the SvIOK and SvNOK macros. These check an SV for a number or an integer, or more precisely, they check to see if the double-value portion of an SV is valid at that point in the code.

Originally, I had the SvIOK and SvNOK macros around any code to which I was expecting to send an integer. The problem is this will not accept code like the following,

```
sc_put_num_val("34.3"); # this is in perl
```

because it is being passed a string value and the number part of the SV was not valid at that time. The SvIV and SvNV macros will convert this to a number even if it is not a valid string. I was parsing strings from a file using regular expressions, and the value I would get in $1 would be a string, even though it was numeric. Once I realized SvNV would produce a number for me, my test script started working.

Summary

This example is not the cleanest implementation of embedding Perl into an application. It was meant as a quick solution to a problem. With an embedded Perl interpreter, sc is quite a bit more powerful than before. One example included with the source is a mortgage calculator that grabs the interest rates form the CNN Financial News web site. With all the Perl modules available, the possibilities are endless.

Resources

John Quillan is a Software Engineer in the Phoenix area. He does tool smithing. When not at work, he enjoys mountain biking, spending time with his girlfriend Niki (Ohh) and programming Perl under Linux. He can be reached at quillan@doitnow.com.

Archive Index Issue Table of Contents

Advanced search

LJ Interviews Guido van Rossum

Andrew Kuchling

Issue #55, November 1998

Mr. Kuchling talks to the creator of Python to find out about the past, present and future of this versatile programming language.

Guido van Rossum is the author of the Python interpreted language. I interviewed him at the end of July to find out what's been happening with Python, and what can be expected in its future.

The Past

Andrew: What inspired you to write the Python interpreter?

Guido: One, I was working at Centrum voor Wiskunde en Informatica (CWI) as a programmer on the Amoeba distributed operating system, and I was thinking that an interpreted language would be useful for writing and testing administration scripts. Second, I had previously worked on the ABC project, which had developed a programming language intended for non-technical users. I still had some interesting ideas left over from ABC and wanted to use them.

I had a two-week Christmas holiday with nothing to do. So, I wrote the first bits of the Python interpreter on my Mac so that I didn't have to log into CWI's computers.

Andrew: What other languages or systems have influenced Python's design?

Guido: There have been many. ABC was a major influence, of course, since I had been working on it at CWI. It inspired the use of indentation to delimit blocks, which are the high-level types and parts of object implementation. I'd spent a summer at DEC's Systems Research Center, where I was introduced to Modula-2+; the Modula-3 final report was being written there at about the same time. What I learned there showed up in Python's exception handling, modules, and the fact that methods explicitly contain “self” in their parameter list. String slicing came from Algol-68 and Icon.

C is a second influence, second only to ABC in importance. Most of Python's keywords, such as **break**, **continue** and others, are identical to C's, as are operator priorities. C also affected early extension modules; many of them, such as the socket and POSIX modules, are simply the corresponding UNIX C functions translated into Python, with some modifications to make programming more comfortable. For example, errors raise exceptions rather than return a negative value, and sockets are objects.

The Bourne shell was also a model for Python's behaviour. Like the shell, Python can execute scripts by specifying their file name, but when run without arguments, it presents you with an interactive prompt. This is well suited for experimenting with the language or with a new module you're trying to learn.

The Present

Andrew: What are the most interesting Python applications you've seen?

Guido: People are doing many neat things with it. Infoseek uses Python as part of their Ultraseek search engine; the web crawler portion is written in Python and can be configured by writing Python code. A group at Lawrence Livermore National Labs is using Python to control numerical calculations. Companies such as Digital Creations are using Python in web-related products.

Andrew: What features of Python are you most pleased with?

Guido: The feel of the whole system suits my style of programming well, for obvious reasons. The ability to run the interpreter interactively and the ability to write code from the bottom up and test it piecemeal combine to let me write code quickly. Other people find that it makes them more productive, too.

Python also connects well to the environment where it's running. ABC was monolithic, putting its own abstractions on top of the operating system. For example, ABC had persistent name spaces that carried the values of variables across successive executions. These name spaces were implemented as disk files, but there was no way to read an arbitrary file or to wander around the file system. I thought that was a serious mistake, so extensibility was important from the start with Python, and it's not difficult to write an extension to interface to a new C library.

Andrew and Guido discuss the future of Python

Andrew: How has the user community surprised you? How did it influence the language's direction, once it started to be used widely?

Guido: Once the source code was made available for downloading, it forced me to make the installation process very easy, just to save myself from answering the same questions over and over. It also forced me to write better documentation.

Much of the language still consists of what I like, but users have always been important in suggesting new things and in fine-tuning them. Some of the early adopters of the language, such as Tim Peters and Steve Majewski, focused on very subtle design details and helped immensely by clarifying the way various features should work; e.g., they convinced me to support mixed arithmetic. Much of the current (and still growing) set of Internet-related modules was contributed or suggested by users.

Andrew: What feature of Python are you least pleased with?

Guido: Sometimes I've been too quick in accepting contributions, and later realized that it was a mistake. One example would be some of the functional programming features, such as lambda functions. **lambda** is a keyword that lets

you create a small anonymous function; built-in functions such as **map**, **filter** and **reduce** run a function over a sequence type, such as a list.

In practice, it didn't turn out that well. Python has only two scopes: local and global. This makes writing lambda functions painful, because you often want to access variables in the scope where the lambda was defined, but you can't because of the two scopes. There's a way around this, but it's something of a kludge. Often it seems much easier in Python just to use a **for** loop instead of messing around with lambda functions. **map** and friends work well only when a built-in function that does what you want already exists.

Andrew: In your introduction to Mark Lutz's book Programming Python (O'Reilly, 1996), you mention that your day job involves writing Python code. What is your job, and how do you use Python in it?

Guido: I now work at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia. CNRI was interested in building a mobile code system called the Knowbot System, because mobile programs are useful for many different purposes. For example, let's say you're indexing a web site. Normally, you'd download all the data using the network and index it, which uses a lot of bandwidth. Also, you can't detect whether two documents are identical copies of each other without downloading them, causing even more wasted time and resources. With mobile code, you could run the indexing program on the server sending only the index, which is a relatively small amount of data, back when the job is done.

CNRI realized that mobile agents would have to be executed in a restricted environment to prevent them from damaging anything on the system; therefore, running Knowbot code in an interpreted language would be required. So, CNRI looked at the languages available at the time, settled on Python as the overall best language for the job and hired me to work on the Knowbot system.

Since then we've actually built the Knowbot system, which required relatively few changes to Python; most of the pieces for restricted execution were already in place even before I started at CNRI. Along the way, CNRI also developed Grail, a web browser written in Python, and most recently we've developed a load-balancing system as part of another research effort. Another application for Knowbots, a subject for future research, is protection of intellectual property. To prevent people from making illegal copies of documents purchased in electronic form, the documents might be embedded inside a Knowbot program that verified the user was authorized to view them. How to make this work is still an open question!

The Future

Andrew: What ongoing Python-related developments do you find most exciting?

Guido: We're working on a Python consortium that would fund future development of Python, but there's nothing to report at this time (late July 1998). It's very exciting to see Python find its way into ever more new products and projects. If I had to name one really big exciting thing, it would be JPython.

Andrew: What exactly is JPython, and what is it suited for?

Guido: The original Python interpreter is written in C. JPython is a completely new implementation of Python written in Java. It brings to Java much of the same advantages that CPython has for a C environment—the interactivity, the high-level language, the ability to glue components together. It can also compile Python programs into Java byte code, which can then be used by other Java code, run as an applet or whatever.

Jim Hugunin has done an excellent job of writing JPython, and it integrates with Java very well. In CPython, to use a new C library you need to write an extension module for it. While there are tools which help with this task, such as David Beazley's SWIG, it still takes some work. Java has the Reflection API, which is an interface for getting information about classes, functions and their arguments. JPython uses the Reflection API to automatically generate an interface, so it can call any Java class without effort.

Andrew: Can you say something about any new features planned for future versions of the interpreter?

Guido: Unicode support is becoming important, and JPython already uses Unicode for its string type, so something will have to be done about that for CPython. The Tk interface could be improved in various ways—speed optimizations, mostly. I'm also thinking about adding Numeric Python's array type to the core language; that presents a few issues, because NumPy's arrays don't behave quite like standard Python lists in various ways, and it might be confusing. Another topic of interest is removing the distinction between classes implemented in Python and extension types implemented as C modules. JPython has a better model for this than CPython does, so those improvements may propagate back into CPython from JPython.

Several proposals have been made for interesting features that would be quite incompatible with the current interpreter, so I'm not sure what should be done about them. For example, one suggestion is static typing; a given variable could be declared to be an integer or a string. That ability would let us catch more

errors at compile time, and would let JPython produce a better translation to Java byte codes. We're still thinking about how to implement that one.

Andrew: What things would you like to see for Python?

Guido: Better database access modules, an integrated development environment, more documentation and more users, of course!

Andrew Kuchling works as a web site developer for Magnet Interactive in Washington, D.C. One of his past projects was a sizable commercial site that was implemented using Python on top of an Illustra database. He can be reached via e-mail at akuchling@acm.org.

Archive Index Issue Table of Contents

Advanced search

The Python HTMLgen Module

Michael Hamilton

Issue #55, November 1998

Mr. Hamilton tells us how to use HTMLgen, a Python-class library, for generating HTML.

This article is about using HTMLgen, a Python-class library for generating HTML. Python is an object-oriented scripting language that ships with most Linux distributions. It plays a major role in configuration and management for distributions such as Caldera and Red Hat. HTMLgen is an add-on Python module written by Robin Friedrich, and available from http://starship.python.net/lib.html under a BSD-style freeware license.

HTMLgen provides classes to support all the standard HTML 3.2 tags and attributes. It can be used in any situation where you need to dynamically generate HTML. For example, you might want to format the results of a database query into an HTML table, or generate an HTML order form customized for each client.

I'll introduce HTMLgen by using it to format data found on typical Linux systems. I think the examples are sufficiently straightforward that they can be followed by anyone familiar with HTML and scripting, and without prior knowledge of Python. Just remember that in Python, blocks of statements are indicated by indenting the code—there are no begin/end statements and no curly braces. (In Python, WYSIWYG applies.) Other than this, Python code looks much like that found in any mainstream programming language.

Although Perl is the most commonly used web scripting language, I personally prefer Python. It can achieve results similar to Perl, and I think Python's syntax, coupled with the style established by its user community, leads to a cleaner, simpler style of coding. This is an advantage during both development and maintenance. These same strengths provide a gentler learning curve for new players. Python moves a little away from traditional scripting languages and more toward non-scripting, procedural programming languages. This allows

Python scripting to scale well. When a small set of scripts starts to grow to the size of a full-blown application system, the language will support the transition.

Getting Started

Any Python program needing HTMLgen must import it as a module. Starting from **bash**, here's how I set up and import HTMLgen to create a “Hello World” web page:

```
bash$ export PYTHONPATH=/local/HTMLgen:$PYTHONPATH
bash$ python
 >>> import HTMLgen
 >>> doc = HTMLgen.SimpleDocument(title="Hello")
 >>> doc.append(HTMLgen.Heading(1, "Hello World"))
 >>> print doc
```

First, I set the PYTHONPATH to include the directory where the HTMLgen.py module can be found. Then, I start the Python interpreter and use its command-line interface to import the HTMLgen module. I create a document object called doc and add a heading to it.

Finally, I print the doc object which dumps the following HTML to standard output:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<!-- This file generated using HTMLgen module.
-->
<HEAD>
 <META NAME="GENERATOR" CONTENT="HTMLgen 2.0.6">
 <TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H1>Hello World</H1>
</BODY> </HTML>
```

Figure 1. Table—Code in Listing 1

This is a start, although not an exciting one. HTMLgen is a very good tool for generating HTML tables and lists. The table in Figure 1 was created by the Python script in Listing 1. The data in the table comes from the Linux /proc/interrupts file which details the IRQ interrupts for your Linux PC. On my PC, doing a **cat** of /proc/interrupts yields:

```
0: 2348528 timer
1: 42481 keyboard
2: 0 cascade
3: 47735 + serial
4: 75428 + serial
5: 48 soundblaster
8: 0 + rtc
11: 1 NE2000
13: 1 math error
14: 175816 + ide0
15: 216 + ide1
```

The Python script reads the contents of the /proc/interrupts file and copies the data into an HTML table. I'll describe this process step by step. As in the previous example, I first create a simple document. I then add an HTMLgen table to the document:

```
table = HTMLgen.Table( tabletitle='Interrupts',
 border=2, width=100, cell_align"right",
 heading=[ "Description", "IRQ",
"Count" ])
 doc.append(table)
```

When creating the table object, I set some optional attributes by supplying them as named arguments. The final headings argument sets the list of column headings that HTMLgen will use. All of the above arguments are optional.

Once I've set up my table, I open the /proc/interrupts file and use the **readlines** method to read in its entire contents. I use a **for** loop to step through the lines returned and turn them into table rows. Inside the loop, the string and regular expressions functions are used to strip off leading spaces and split up each line into a list of three data values based on space and colon (:) separators:

```
data=regsub.split(string.strip(line),'[ :+]+')
```

Elements of the data list are processed to form a table row by reordering them into a new three-element list consisting of name, number and total calls:

```
[ HTMLgen.Text(data[2]), data[0], data[1] ]
```

The outer enclosing square brackets construct a list out of the comma-separated arguments. The first list element, **data[2]**, is the interrupt name. The interrupt name is a non-numeric field, so I've taken the precaution of escaping any characters that might be special to HTML by passing it though the HTMLgen Text filter. The resulting list is made into a row of the table by appending the list to the table's body:

```
table.body.append(
        [ HTMLgen.Text(data[2]), data[0], data[1] ])
```

Finally, once all lines have been processed, the document is written to interrupts.html. The result is shown in Figure 1.

The simple Table class is designed for displaying rows of data such as might be returned from a database query. For more sophisticated tables, the TableLite object offers a lower-level table construction facility that includes the ability to do individual row/column customization, column/row spanning and nested tables.

Figure 2. Bar Chart—Code in Listing 2

Bar Charts from HTML Tables

The table facilities have also been extended to create fancy bar charts. Figure 2 shows a bar chart I generated from the output of the Linux **ps** command. The chart was created by the HTMLgen bar-chart module. The code for psv.py is the 20 lines of Python code shown in Listing 2. The original output from **ps v** looks something like the following:

```
PID TTY STAT TIME PAGEIN TSIZ DSIZ RSS LIM %MEM COMMA
 555 p1 S 0:01 232 237 1166 664 xx 2.1 -tcsh
1249 p2 S 0:00 424 514 2613 1676 xx 5.4 xv ps
 ...
```

I use the Python operating system module's **popen** function to return a file input pipe for the output stream from the command:

```
inpipe = os.popen("ps vax", "r");
```

I then read in the first line from the input pipe and split it into a list of column names.

```
colnames = string.split(inpipe.readline())
```

Now, I create the chart object, and the chart object's datalist object:

```
chart = barchart.StackedBarChart()
...
chart.datalist = barchart.DataList()
```

Datalists can have multiple data segments per bar, which results in a stacked bar chart such as the one in Figure 2. I need to tell the datalist object how many data segments are present by setting the list of **segment_names**. I decided the bars on my chart will have two segments, one for TSIZ (program text memory size) and one for DSIZ (program data memory size). To accomplish this, I need to copy the two column names from **colnames** into **segment_names**. Because lists in Python are numbered from zero, the two **colnames** I'm interested in are columns 5 (TSIZ) and 6 (DSIZ). I can extract them from the **colnames** list with a single slicing statement:

```
chart.datalist.segment_names = colnames[5:7]
data = chart.datalist
```

The **[5:7]** notation is a slicing notation. In Python, you can slice single items and ranges of items out of strings, lists and other sequence data types. The notation **[low:high]** means slice out a new list from low to high minus 1. On the second line, I assign the variable called “data” to the variable “chart.datalist” to shorten the length of the following lines to fit the column width required in *Linux Journal*.

After initializing the chart, I use a **for** loop to read the remaining lines from the **ps** output pipe. I extract the columns I need by using **string.split(line)** to break

the line into a list of columns. I extract the text of each command by taking all the words from column 10 onward and joining them into a new **barname** string:

```
barname = string.join(cols[10:], " " )
```

I use the string module's **atoi** function to convert the ASCII strings in the numeric fields to integers. The last statement in the loop assembles the data into a tuple:

```
( barname, tsize, dsize )
```

A tuple is a Python structure much like a list, except that a tuple is immutable—you cannot insert or delete elements from a tuple. Although the two are similar, their differences lead to quite different implementation efficiencies. Python has both a tuple and a list, because this allows the programmer to choose the one most appropriate to the situation. Many features of Python and its modules are designed to be high-level interfaces to services that are then implemented efficiently in compiled languages such as C. This allows Python to be used for computer graphics programming using OpenGL and for numerical programming using fast numerical libraries.

Back to the example. The last statement in the loop inserts the tuple into the chart's datalist.

```
data.load_tuple(( barname, tsize, dsize ))
```

When the last line is processed, the loop terminates and I sort the data in decreasing order of TSIZ:

```
data.sort(key=colnames[5], direction="decreasing")
```

After that, I create the final document and save it to a file.

```
doc = HTMLgen.SimpleDocument(title='Memory')
doc.append(chart)
doc.write("psv.html")
```

Loading psv.html into a browser results in the chart seen in Figure 2. By altering the bar chart's parameters, such as the GIFs used for the chart's “atoms”, I can build the chart in different styles.

Figure 3. HTML Page Generated by Listing 3

Multiple Documents

In my next example, I'll show you how a data stream can be processed to produce a series of documents that are interlinked. The script in Listing 3

creates a set of two documents summarizing all the Red Hat packages installed on a Linux system. The resulting HTML page is shown in Figure 3. An index document summarizes the RPM major groups, and a second main document summarizes the RPMs in each group. A link for each group in the index document jumps directly to each group's entries in the main document.

The HTML is generated from the output of the following **rpm** command:

```
rpm -q -a --queryformat \
   '%{group} %{name} %{summary}\n'
```

The output typically looks like this:

```
System/Base sh-utils GNU shell utilities.
Browser/WWW lynx tty WWW browser
Programming/Tools make GNU Make.
System/Library xpm X11 Pixmap Library
System/Shell pdksh Public Domain Korn Shell
```

I read this output into a Python list and pre-process it by sorting it into alphabetic order.

I produce two documents, an index document (**idoc**), and a main document (**mdoc**), using HTMLgen's SeriesDocument to give both documents the same look-and-feel. By using a SeriesDocument, I can configure standard document headers, footers and navigation buttons via an **rcfile** and other optional arguments.

The index document (**idoc**) has only one HTMLgen component: an HTML list of RPM groups. I've used the **HTMLgen.List** columns option to create a multi-column list:

```
ilist = HTMLgen.List(style="compact", columns=3)
idoc.append(ilist)
```

The **for** loop processes each line from the **rpm** command and generates both **idoc** and **mdoc** text. Each time the group name changes, I add a new list entry to the ilist:

```
if group != lastgroup:
 lastgroup = group
 title = HTMLgen.Text(group)
 href = HTMLgen.Href(mainfile+"#"+ group,
        title)
 index.append(href)
```

I've wrapped the list text in an HTML-named HREF, linking it back into **mdoc**. I've used the name of the main file and group title to form the HREF link. For example, in the case of the “Browser/FTP” RPM group, my code would generate the following HREF link:

```
<A HREF="rpmlist.html#Browser/FTP">Browser/FTP</A>
```

The main document (**mdoc**) has a more complex structure. It consists of a series of HTML definition lists, one per RPM group. Each time the group name changes, I generate the named anchor that is the target for the reference generated above:

```
anchor = HTMLgen.Name(group, title)
```

I append the anchor to **mdoc** as a new group heading:

```
mdoc.append(HTMLgen.Heading(2, anchor))
```

For the “Browser/FTP” group, this would generate the following HTML:

```
<H2><A NAME="Browser/FTP">Browser/FTP</A></H2>
```

Once the group heading has been appended, I start a list of RPMs in the group:

```
grplist = HTMLgen.DefinitionList()
```

Once a new group list has been started, my **for** loop will keep appending RPM summaries to the **mdoc** until the next change in group name occurs:

```
grplist.append(
 (HTMLgen.Text(name),HTMLgen.Text(summary)))
```

When the entire rpmlist has been processed, I generate the two documents you see in Figure 3.

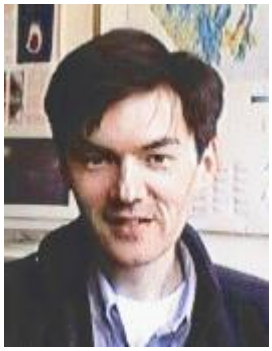
In this example, I simultaneously generated two simple documents and linked one to the other. This example could easily be extended to provide further links to individual documents for each RPM, and from each RPM to the RPMs it depends on.

Where To From Here

I've only scratched the surface of what's possible with HTMLgen and Python. I haven't covered the HTMLgen objects for HTML Forms, Image Maps, Nested Tables, Frames, or Netscape Scripts. I also haven't made use of Python's object-oriented nature. For example, I could have sub-classed some of the HTMLgen objects to customize them for specifics of each application. I haven't discussed the Python module for CGI handling. You can read more about these topics by pointing your browser at some of the references accompanying the article (see Resources).

If you're trying to get started with HTMLgen, the HTMLtest.py file distributed with HTMLgen provides some good examples. The HTMLgen documentation is quite good, although in some cases, more examples would help. I don't think my examples require any particular distribution of Linux, libc or Python. All of them were written using HTMLgen 2.0 with Python 1.4 on Caldera OpenLinux Standard version 1.2.

Resources



Michael Hamilton is a freelance UNIX C/C++ developer. Recently he's been working on Web and Java projects, as well as C++ fault-tolerant UNIX applications. Michael tripped over one of Linus's postings back at the beginning of 1992 and has been hooked ever since. Michael currently runs two Linux hosts, a main workstation and an old 386 used as an Xterminal. Both of these systems have been put to use on projects to be delivered on Linux and other UNIX platforms. He can be reached at michael@actrix.gen.nz.

Archive Index Issue Table of Contents

Advanced search

Xforms Marries Perl

Reza Naima

Issue #55, November 1998

How to add a powerful graphical user interface to Perl scripts

As you have seen in various recent issues of *Linux Journal*, the Xforms library allows you to add a powerful GUI to your C or C++ programs using a simple, intuitive API. The functionality and elegance of Xforms GUI is comparable to Motif's, yet the Xforms libraries are free if used non-commercially. Thanks to Martin Bartlett (martin@nitram.demon.co.uk), the Xforms' GUI can be used from Perl scripts to run complicated graphical applications or to provide a simple “please wait while loading” status bar so the user doesn't get bored waiting (see Figure 1). This article will discuss how to install Xforms4Perl (version 0.5) and how to write a simple address book program with it.

Figure 1. Status Bar

Installing Xforms4Perl

In order to install Xforms4Perl, first you must have installed the following:

- Perl version 5.003 (or higher), compiled to load libraries dynamically.
- XForms Library, version 0.86 or 0.88, which can be found at http://bragg.phys.uwm.edu/xforms.

Next you need to obtain the source code for Xforms4Perl, which you can do from either the author's primary site, ftp://ftp.demon.co.uk/pub/perl/perl/, or any of the CPAN mirror sites under the directory /authors/Martin_Bartlett/. You can also get an RPM from ftp://ftp.redhat.com/pub/contrib or CPAN, and skip the next few sections.

Once you have Xforms4Perl downloaded, unpack it into a convenient directory (i.e., /usr/local/src) using **tar zxvf Xforms4Perl-0.8.4tgz**. This command creates the subdirectory Xforms4Perl-0.8.4.

Xforms allows for the support of OpenGL, and if you want to access it from Xforms4Perl, or if you need to modify some default paths or library locations, you will need to edit the Makefile.PL files located in the subdirectories X11/XEvent, X11/XFontStruct and X11/Xforms.

Then enter the X11 subdirectory and do the following:

- Type **perl Makefile.PL**.
- Type **make**.
- Type **make install** (as root).
- Copy **fd2Perl** to a directory that is in your PATH.

Once installation is complete, you can then start writing Perl code which uses Xforms. You might also check out some of the demos that come with PerlXForms, such as the author's **XFtool** which is similar to the Microsoft Office Toolbar. The rest of this article assumes you have an existing fundamental understanding of Xforms and Perl, although both are so easy to use you can probably pick them up from looking at a few examples.

Constructing the Look of the Application

In order to help explain how to use the Xforms4Perl Library, I will use as an example the development of a simple phone book application from start to finish. The best place to start is with the **fdesign** application that comes with the Xforms library. It allows you to build the components of your application visually. Rather than trying to figure out if your button should be 33 by 55 pixels or 30 by 50 pixels, you just draw it how you want it, and fdesign deals with all the numbers and details. In order to make things even easier, fdesign is able to output Perl code (thanks to the fd2Perl script mentioned above). We invoke fdesign as **fdesign -perl**, and create a new form called “list” (see Figure 2).

Figure 2. Form Design

First, we add an object called **browser** where the phone book name entries get indexed. Under the attributes section, we specify that we want this to be a **HOLD_BROWSER**, which allows a selection from the browser to remain highlighted after selection. Then we give it an obvious name, such as **browser**, and set up a callback function. This function will be executed when some action takes place in the browser—using callback functions is fairly standard in programming GUIs. We randomly pick the name **browser_clicked** for the callback function.

We now add five text input fields, all with the same callback function, named **data_change**. These fields will display the personal information from the phone

book entries and are also the locations where the user can make data modifications. These fields are labeled and given the following names in the attributes section:

Label	Name
Name	name_field
Phone Number	phone_field
Address	address1_field
Address	address2_field
E-Mail	email_field

Next, we add four buttons. A pull-down menu could have been used here, but four buttons shouldn't clutter the interface and will be easier to access than a menu. The buttons are labeled “Quit!”, “Clear”, “Update” and “Delete Entry”. The purpose of the Quit button should be obvious. The Clear button is used to clear the text input fields, the Update button is used to save or update whatever is in the text input fields, and the Delete Entry button is used to remove the selected entry from the browser listing. These buttons each have a callback, as listed below.

Label	Callback Function
Quit!	quit
Clear	clear_data
Update	update_data
Delete Entry	delete_entry

Finally, we can add a title such as “PhoneBook” or anything else to improve the appearance, which is quite simple to do using fdesign—you just place it and you're done. I also thought it might be nice to give some of the buttons a shadow effect, which is done from the respective button's attributes menu.

Going from the Picture to the Perl Code

Now we have finished designing the application, and we can save it as a file called pbook. This operation generates four (or more, based on how you set it up) files. The one of interest to us is called pbook.pl (Listing 1). If we ever want to modify our design, the .pl file is regenerated; thus, we should not modify this file, as any changes to it would be lost. Rather, in our code, we need to *require* the file to set up our form for us.

The Essentials of the Code

For a Perl script to use Xforms4Perl, all you have to do is add the line:

```
use X11::Xforms;
```

to the top of your Perl script. This command dynamically loads everything that needs to be loaded. If you want to access the XFontStruct or XEvent structures, then you will also need to add **use X11::XFontStruct** or **use X11::XEvent**. Before doing anything involving the Xforms library, you must call a function so that Xforms can initialize all of its internal variables, etc. This is done by calling

```
fl_initialize('Phone Book');
```

and giving it the title of the application you want. You set up all the forms you need by making calls to subroutines, such as:

```
create_form_list();
```

which have been generated by fdesign and fd2perl. Then, add whatever initialization code you need. For example, if you make a “slider” using fdesign and need to set the range of values of the slider based on some input, you will need to call **fl_set_slider_bounds** with the proper parameters. After you have finished configuration, you're ready to show the form to the user. To accomplish this, make the call:

```
fl_show_form($list,FL_PLACE_FREE,FL_FULLBORDER,
"Phone Book");
```

There are also commands to make the form disappear, so it's fairly easy to handle multiple forms. The options you feed **fl_show_form** include the specific form you wish to display (remember we called ours “list”), options for the window manager and a title. Once everything is set up and you're ready for the user to interact with the GUI, you then keep calling the function **fl_do_forms**. Thus, the minimal amount of code you need to display this form is:

```
#!/usr/local/bin/perl
use X11::Xforms;
require "pbook.pl";
fl_initialize('Phone Book');
create_form_list();
fl_show_form($list, FL_PLACE_FREE, FL_FULLBORDER,
```

```
                                    'Phone Book');
while(1) {fl_do_forms};
```

The actual interaction the user has with the GUI is accomplished via callback functions. Keep in mind this represents the bare essentials of a basic mode of using Xforms4Perl, and there are many other ways of using the libraries. The Xforms manual should be consulted for more advanced methods.

Back to the Phone Book

Let's look at how our phone book program is now pieced together. A full listing can be found in Listing 2. First, look at the entire main routine. In addition to the above listing, we've added the calls to two subroutines to initialize the browser before displaying it with the **fl_show_form** function.

```
load_data();
update_browser()
```

Figure 3. Phone Book

As you can see, most of the program is handled in subroutines. First, we load the libraries and initialize the program. The call to **create_form_list** sets up the form we pieced together using fdesign. The call to **load_data** loads the data from a file in the user's home directory called .pbook. The format for this file has been arbitrarily set up to contain one phone book entry per line with the data delimited by colons. (Assume the user would never use a colon in any entry. See Figure 3.) A sample entry (one line) from .pbook would look like this:

```
Suhad:414 555-1234:2014 S 102nd St:Milwaukee,
WI 53227:sniazi@ucsd.edu
```

Now that the data is loaded, the **update_browser** call clears the browser and adds the newly loaded names to the browser list. To show the phone book to the user, we make the call to **fl_show_form**, and then loop through **fl_do_forms** to let Xforms take over.

The rest of the interaction with the user is handled through callbacks, which include adding or updating entries, saving the new entries, displaying information and quitting the application. The complete code is shown in Listing 2, and it's fairly evident how the callbacks work together, given Xforms' verbose function names. For example, **fl_get_input** gets the data from an input field, and **fl_set_input** sets it to some value. To get a better feeling for all the functions available, you can print out the 200-page manual that comes with Xforms; it makes an invaluable reference.

Although it should be fairly easy to guess what the code in Listing 2 does, there are a few things I'd like to mention and clarify. First of all, when the data is

loaded, it is kept in a data structure called **$DATA** which is a pointer to an anonymous hash containing the names of the people in the address book. In turn, each of those names is a pointer to another anonymous hash containing the phone number, address, etc. For example, to access Tom's phone number, you would use **$DATA->{'Tom'}->{pnumber}**. Because we chose to handle the storage and organization of data using this method, we run into a slight problem. If we modify the name of an entry (i.e., change Tom to Tommy), the program thinks it is dealing with a new entry (because the hash's key is changed), and thus, two entries are made: one based on the name Tom and one based on the name Tommy. I leave it as an exercise for the reader to find a solution for this problem.

Most subroutines use either the **fl_set_input** or **fl_get_input** calls to manipulate the data you see. The only complicated call is

```
$name=fl_get_browser_line($browser,
fl_get_browser($browser));
```

This call is used to determine the highlighted name from the browser. For example, in Figure 2, the highlighted name is Suhad. In order to do that, we need to give **fl_get_browser** the line number in which the entry appears. To get that number, we make a call to **fl_get_browser_line** which returns with the current highlighted line number.

Another thing to note is that rather than having a “save” button, this phone book assumes you wouldn't make a mistake. Thus, it saves the data every time you click on either the Update button or the Delete Entry button. A backup of the last saved file is kept as .pbook.bak in case of a problem, but the user has to restore the backup himself—it's not automatic.

All listings referred to in this article are available by anonymous download in the file ftp.linuxjournal.com/pub/lj/listings/issue55/2024.tgz.

Reza Naima (reza@reza.net) spent the last year working at Cisco Systems in charge of web server security. Among his duties were ensuring any deployed code was secure, developing security standards and writing automation applications. He just left Cisco to go on a 3.5 month trip around the world, returning in mid-September.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

The Quick Road to an Intranet Web Server

Russell C. Pavlicek

Issue #55, November 1998

Apache and Linux make the task simple.

On a recent project, I was busily working on my Linux workstation while a nearby coworker was attempting to install a well-known web server on a popular operating system. As the day wore on, his frustration grew. By the end of the day, he had most of the web server functioning, but was still experiencing difficulty with configuring certain portions of it.

Here was a man of proven technical abilities fighting to set up an Intranet web server for a project. It obviously wasn't a simple task. Through dogged determination, my friend eventually prevailed, but not without gaining a few gray hairs in the process.

I began to reflect on my own experience with web servers. The unobtrusive Linux workstation I was using (a 486/66 with 16 MB of memory and a 0.5 GB SCSI drive) was equipped with the Apache web server. Six months earlier, I had built this system from unused parts lying around our lab and installed Red Hat 4.2 on it. When I built the machine, it was my hope to demonstrate to my coworkers how impressive and substantial Linux had become.

What I hadn't expected was to find myself even more impressed with Linux and Apache as a result of my friend's experience.

When I installed Red Hat 4.2 on the machine, I simply specified that I wanted the Apache web server installed along with the other packages I selected. When the installation was complete, I simply aimed a browser at the newly installed machine and found a friendly little Red Hat-supplied web page staring at me, telling me which file to edit to begin loading content into my new web site.

No fussing with parameters or complex configuration was needed. I just began editing the default HTML file, and soon had a neat web site exemplifying the benefits of Linux.

Why an Intranet Web Server?

As corporations worldwide try to catch the Internet wave, more and more companies realize the need to build a decent Intranet in order to share information within their organization. Traditional paper documents, such as policy manuals, software documentation, design specifications, press releases and corporate announcements, are suddenly more accessible within the organization via an Intranet web server. New capabilities such as support discussions, document searches and audio and video archives are creating opportunities to increase the availability of information and to add to an organization's competitive advantage.

At the center of many of these technologies is the need for a robust and efficient Intranet web server. Yet even a casual glance at the commercial software solutions employed for developing an effective server will reveal that constructing one can be an expensive proposition.

Here is a place where Linux and Apache shine! The combination of the robust Linux operating system with the industry-leading Apache web server (see Resources) creates a flexible foundation for a low-cost, highly functional Intranet web site. Using Linux to solve a business problem is one of the best methods of convincing people that this operating system can hold its own in the corporate arena.

Of course, Linux and Apache can also be configured as a highly effective Internet web server. An Internet server requires all the elements of a good Intranet server, plus a good deal more. In particular, security and performance are usually much more critical in an Internet scenario. However, the issue of installing a web server into a less hostile internal network can be handled with ease, especially if you are using a Linux distribution which does most of the work for you.

What if your Linux distribution doesn't come with Apache preconfigured? The good news is that it doesn't take much to install Apache on most any Linux machine. It is possible to set up a working web server on your Intranet in just a few minutes without having to be a technical wizard.

How to Install

Installation of Apache is a snap using Red Hat 4.2, Debian 1.3.1, or OpenLinux Base 1.x. The Debian distribution goes through a short configuration dialogue

(if in doubt, just press the ENTER key a few times), while the Red Hat and OpenLinux distributions have a more or less preconfigured installation. In all three distributions, the process of creating a working Intranet server takes neither experience nor a significant amount of time.

If your distribution doesn't have an Apache package, you can always obtain a source kit from http://www.apache.org/. Here's an example (using Apache 1.2.5) of the commands used to quickly unpack and build the program:

```
tar xzf apache_1.2.5.tar.gz
cd apache_1.2.5/src/
./Configure
```

If you have a different location to store the configuration files other than the default location of /usr/local/etc/apache/conf/, simply change the following line in src/httpd.h:

```
#define HTTPD_ROOT "/etc/httpd"
```

Note that the definition of **HTTPD_ROOT** does **not** include a closing slash. If you don't make the change in the file, but still want to move the configuration files, you will be able to specify the location of httpd.conf at startup time by using the **-f** option on the **httpd** command line.

Regardless of whether or not you decide to make the change above, the final step is to compile the web server. Just type **make** and the httpd executable will be created in the /src subdirectory. The source kit contains additional instructions in README and src/INSTALL should you wish to do anything fancy. A vanilla compile should work just fine for most situations.

How to Configure

Apache has many wonderful configuration options available. To someone who has never managed a web server, the list of options might seem quite daunting. However, it is possible to employ a simple cookbook method to get your web server up and running in short order. The following values will produce a working Intranet server which should perform marvelously in most organizations. Of course, if you have special security requirements, a review of the Apache documentation is quite helpful.

Any Linux distribution with an easy installation of Apache will likely have a reasonable set of parameters already enabled. Even if your web server is live seconds after installation, you may want to review the configuration files just to find out what features have been enabled and disabled by default.

The Apache configuration generally has four files of note. In Red Hat, these are found in /etc/httpd/conf/. In Debian, they are located in /etc/apache/. If you built Apache from the sources as described above, they should be in /usr/local/etc/apache/conf/. The four files are access.conf, httpd.conf, srm.conf and mime.types.

If the files are not found in your kit, you should be able to locate the same group of files with names ending in .conf-dist. Simply copy each of these files to the appropriate location (e.g., /etc/httpd/conf/ if you are using Red Hat) using the .conf file type, and edit the files so that they contain the parameters described below. Note that the Apache-supplied configuration files contain useful hints not shown here for the sake of space, as well as many additional parameters not discussed here. So, it is best to edit the files to contain the following parameters, rather than attempting to construct new configuration files from scratch.

Of the four files, mime.types is the least likely to require modifications. It is a table which associates MIME headers with file types. For example, the table will say that a file ending with .gz should generate a MIME header of application/x-gzip.

The most likely candidate for parameter adjustment is the access configuration definition. In access.conf, try using the following values:

```
<Directory /home/httpd/html>
Options Indexes Includes ExecCGI FollowSymLinks
AllowOverride None
order allow,deny
allow from all
</Directory>
```

In this example taken from a modified Red Hat file, this entry says that the main HTML files for this server will be stored in /home/httpd/html. The options include:

- **Indexes**: if the user specifies a URL that points to a directory name rather than a file name and that directory does not contain an index file (such as index.html), Apache will display a list of the files contained in the specified directory. If you don't want this behavior, simply omit the “Indexes” keyword from the “Options” line.
- **Includes**: this allows the server to include files as directed.
- **ExecCGI**: if the URL specified is actually a CGI script, this will allow that script to execute. If you are not interested in CGI scripts, don't include the keyword.
- **FollowSymLinks**: Let's say that I've created a symbolic link to my CD drive in this directory. This keyword will instruct Apache to allow access to that

CD as if it were a subdirectory of this directory. This facility can be quite handy if you wish to serve up the contents of a CD or allow access to the HOWTOs which normally reside in a tree such as /usr/doc. One quick symbolic link, and you can allow Apache to serve those files without moving them or creating individual symbolic links. But remember to create symbolic links with care, or else you might find that your web site is serving up documents you never intended to be universally available.

In the above example, the remaining directives translate (roughly) to “don't override the normal access rules”, “evaluate rules to allow access before the rules to deny access” and “allow access from all hosts”. Together, these rules add up to “if it's there and they ask nicely, give it to them”.

The CGI directory entry will generally look more minimal, like:

```
<Directory /home/httpd/cgi-bin>
AllowOverride None
Options None
</Directory>
```

The parameters that define the operational parameters for the Apache daemon in httpd.conf are as follows:

```
HostnameLookups on
```

If **HostnameLookups** is “on”, the server will use DNS to try to determine the user's host name for logging purposes.

```
User nobody
Group nobody
```

User and **Group** determine the access privileges of the remote user. The server will act as if it were a job created by the specified user and group (in this case, “nobody”).

```
ServerAdmin root@localhost
```

ServerAdmin sets the user name and host to receive mail messages that might be generated by the daemon.

```
ServerRoot /etc/httpd
```

ServerRoot sets the base directory for configuration and log files.

```
ErrorLog logs/error_log
TransferLog logs/access_log
RefererLog logs/referer_log
AgentLog logs/agent_log
```

These parameters specify the name and location of various log files. The directories specified are relative to the **ServerRoot** directory. Once you've run Apache for awhile, examine these log files. In them, you'll find information about which pages were accessed when and by whom. You'll even find

information about the page that called your page and the type of browser the user was employing to look at your site.

```
MinSpareServers 5
MaxSpareServers 10
StartServers 5
MaxClients 150
```

These parameters deal with the fact that Apache generates child processes *before* they are needed in order to deal with any sudden increase in incoming traffic. These parameters specify the minimum and maximum number of unused children which should exist at any point in time. They also specify the absolute minimum and maximum number of children which should be available. Using the command **ps ax** will reveal the multiple children which the Apache daemon is currently using.

The file srm.conf contains many different items. It deals largely with the name space of the server as well as how the server responds to requests. Of special interest are the following lines:

```
DocumentRoot /home/httpd/html
Alias /icons/ /home/httpd/icons/
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/
UserDir public_html
```

DocumentRoot specifies the top of the directory tree that contains the pages to be served. **Alias** allows the specified directory to be accessed by a pseudonym, even though it is not under the **DocumentRoot** tree. **ScriptAlias** is like **Alias** except that the directory will contain CGI scripts. **UserDir** specifies the user's subdirectory to look in for any URL that uses a /~username/ specification. For example, if the user's default directory is normally /home/username/, then the user's default HTML directory will be /home/username/public_html/.

Conclusion

The combination of Linux and the Apache web server is a quick and inexpensive way to build a robust Intranet web server. Despite its multiplicity of options, Apache can be quickly configured to serve the needs of most organizations. In fact, most current Linux distributions come with Apache preconfigured and ready for action. If your needs should become more complex, Apache can grow with you to do the job.

Resources

Russell C. Pavlicek is employed by Digital Equipment Corporation as a software consultant serving U.S. Federal Government customers in the Washington, D.C. area. He lives with his lovely wife and wonderful children in rural Maryland where they serve Yeshua and surround themselves with a variety of furry creatures. In his miniscule amounts of spare time, he continues to develop the Corporate Linux Advocate home page at http://www.geocities.com/SiliconValley/Haven/6087/. His opinions are entirely his own (but he will allow you to adopt one or two if you ask nicely). He can be reached at pavlicek@altavista.net.

Archive Index Issue Table of Contents

Advanced search

XML, the eXtensible Markup Language

Andrew Kuchling

Issue #55, November 1998

XML has been attracting a lot of attention recently. This article provides a five-minute overview of XML and explains why it matters to you.

The Standard General Markup Language is about two decades old. SGML was originally designed for processing large documentation sets, but SGML is neither a programming language nor a text formatting language. Instead, it's a *meta*-language that allows defining of customized markup languages. The most famous SGML-based language today is unquestionably HTML.

Because SGML has been around for two decades, many companies offer SGML tools and products and it's firmly entrenched in many high-end document-processing applications. SGML is quite a large language; however, understanding the basics isn't very difficult. It does contain many rarely used features which are harder to understand. Implementing a full SGML parser is difficult, and this has given SGML a reputation for fearsome complexity. This reputation isn't truly deserved, but it's been enough to scare many people away from using it.

XML, then, is a stripped-down version of SGML that sacrifices some power in return for easier understanding and implementation. It's still a meta-language, but many of SGML's lesser-used features and options have been dropped. The XML 1.0 specification is about 40 pages long, and a parser can be implemented with a few weeks of effort.

A Brief Glance

A mark-up language specified using XML looks a lot like HTML:

```
<?xml version="1.0"?>
<!DOCTYPE myth SYSTEM "myth.dtd">
<myth>
 <name lang="latin">Hercules</name>
 <name lang="greek">Herakles</name>
```

```
  <description>Son of Zeus and Alcmena.</description>
  <mortal/>
 </myth>
```

An XML document consists of a single element containing sub-elements which can have further sub-elements inside them. Elements are indicated by tags in the text, consisting of text within angle brackets **<...>**. Two forms of elements are available. An element may contain content between opening and closing tags, as in **<name>Hercules</name>**, which is a **name** element containing the data Hercules. This content may be text data, other XML elements or a mixture of the two. Elements can also be empty, in which case they're represented as a single tag ending with a slash, as in **<mortal/>**, which is an empty **stop** element. This is different from HTML, where empty elements such as **
** or **<IMG>** aren't indicated differently from a non-empty element such as **<H1>**. Also unlike HTML, XML element names are case-sensitive; **mortal** and **Mortal** are two different element types.

Opening and empty tags can also contain attributes, which specify values associated with an element. For example, text such as **<name lang="greek">Herakles</name>**, the **name** element has a **lang** attribute with a value of “greek”. In **<name lang="latin">Hercules</name>**, the attribute's value is “latin”. Another difference from HTML is that quotation marks around an attribute's value are not optional.

The rules for a given XML application are specified with a Document Type Definition (DTD). The DTD carefully lists the allowed element names and how elements can be nested inside each other. The DTD also specifies the attributes which can be defined for each element, their default values, and whether they can be omitted. For example, to make a comparison with HTML, the **LI** element, representing an entry in a list, can occur only inside certain elements which represent lists, such as **OL** or **UL**.

The document-type definition is specified in the DOCTYPE declaration; the above document uses a DTD called “mythology” that I invented for this article. The “mythology” DTD might contain the following declarations:

```
<!ELEMENT myth (name+, description, mortal?)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name lang ( latin | greek ) "latin">
<!ELEMENT description (#PCDATA)>
<!ELEMENT mortal EMPTY>
```

I won't go into every detail of these lines, however, lines beginning with **<! ELEMENT** are element declarations. They declare the element's name and what it can contain. So, the **myth** element must contain one or more **name** elements, followed by a single **description** element, followed by an optional **mortal** element. (+, * and ? have the same meanings as in regular expressions: one or

more, zero or more, and zero or one occurrence.) The **mortal** tag, on the other hand, must always be empty.

The third line declares the **name** element to have an attribute named **lang**; this attribute can have either of the two values “latin” or “greek” and defaults to “latin” if it's not specified.

A validating parser can be given a DTD and a document in order to verify that a given document is valid, i.e., it follows all the DTD's rules. This is quite different from HTML, since web browsers have historically had very forgiving parsers, and so relatively few people make any effort to write valid HTML. This looseness means that code to render HTML text is full of hacks and special cases; hopefully, XML won't fall into the same trap of leniency.

This article doesn't cover all of XML's features—I haven't discussed all the possible attribute types, what entities are or that XML uses Unicode, which enables XML processors to handle data written in practically any alphabet. For the full details of XML's syntax, the one definitive source is the XML 1.0 specification, available on the Web at the World Wide Web Consortium's XML page (see Resources). However, like all specifications, it's quite formal and not intended to be a friendly introduction or a tutorial. Gentler introductions are beginning to appear on the Web and on bookstore shelves.

Why Does XML Matter?

XML will most likely become very common over the next few years. Many new web-related data formats are being drafted as XML DTDs; three examples are MathML for specifying mathematical equations, RDF (Resource Description Format) for classifying and describing information resources, and SMIL for synchronized multimedia. There are also individual efforts to define DTDs for all sorts of applications including genealogical data, electronic data interchange and vector graphics; the list is growing all the time.

XML isn't primarily a competitor to HTML. The World Wide Web Consortium is planning to base the next generation of HTML on XML, but HTML as it currently stands isn't going to disappear anytime soon. Many people have already learned HTML and are happily using it; they don't particularly want or need the ability to create new markups. There are millions of existing HTML documents now on the web, and converting them to XML would take a long time; many documents may never be converted.

XML *is* going to be very significant, and XML support will be very common. The next versions of the Mozilla and Internet Explorer browsers will each support XML and will use it internally in various ways. More and more new data formats are written as XML DTDs. The argument driving this is simple laziness: if XML is

available on every platform, and if its capabilities are suited to the task, then using XML will save time with little effort, which is always a persuasive argument.

In addition, XML will be easily accessible to programmers. James Clark's Expat parser (see Resources) is high-quality code and is freely available under the terms of the Mozilla Public License. I wouldn't be surprised to see future Linux distributions coming with Expat as part of the base system. Interfaces to Expat for scripting languages such as Python, Perl and Tcl are already in development and will probably have been finished by the time you read this. Soon, adding XML parsing to a program will be as easy as adding **from xml import parsers** or **use XML::Parser** to your code.

Andrew Kuchling works as a web site developer for Magnet Interactive in Washington, D.C. One of his past projects was a sizable commercial site that was implemented using Python on top of an Illustra database. He can be reached via e-mail at akuchling@acm.org.

Archive Index Issue Table of Contents

Advanced search

More Flexible Formatting with SGMLtools

Cees de Groot

Issue #55, November 1998

A brief overview of the latest SGMLtools is presented by one of its developers.

In the October 1995 issue of *LJ*, Christian Schwarz presented a short overview of Linuxdoc-SGML as it stood then: a complete, out-of-the-box package that gave and still gives authors a chance to write once and present anywhere. From flat ASCII to typeset PostScript and hypertext HTML, it all rolls out from a single SGML source file. Since then, lots of smaller and bigger changes have resulted in renaming it SGML-Tools (and then SGMLtools—the hyphen caused confusion) to indicate it wasn't just for Linux anymore. Still, we, the SGMLtools project authors, weren't satisfied with this, so we set out to build an even better package that is presented here, SGMLtools 2. This article will give a brief overview of what happened to SGML-Tools 1 that led us to rename it SGMLtools 2; more extensive information can be found on the SGMLtools web site (see Resources).

From Linuxdoc to DocBook

A big issue that came up again and again was the fact that the shortcomings of the Linux document type definition were beginning to show. Document type definition (DTD) is the SGML term for the set of rules that fixes how an SGML document that is compliant with DTD must look. It outlines the structure of the document from titles and subtitles to tables; everything is defined.

Maintaining a document type definition, as we found out, is quite difficult. Constant discussion took place over which features should be allowed, how to make existing features better, whether to stick with pure procedural markup or be a little bit pragmatic about things. Endless rounds of talks came up and came back and began to interfere with progress. The Linuxdoc DTD was clearly too limited, but we didn't want to redesign it without finding out whether alternatives already existed.

We quickly came to the conclusion that the DocBook DTD, as developed by the Davenport Group, would be a good successor to the Linuxdoc DTD. DocBook, being developed by professionals for professionals with an emphasis towards technical documentation, fits the target audience for SGMLtools very well and solves a number of the problems of Linuxdoc. Furthermore, almost every SGML vendor supports DocBook, so this would make users less dependent on us and give them more ways to process SGML documentation. Recently, responsibility for maintaining DocBook has been transferred to the Organisation for the Advancement of Structured Information Standards (http://www.oasis-open.org/), ensuring that DocBook will continue to be widely supported.

From Mapping Files to DSSSL

The acronym DSSSL may not say much to the average reader, but it stands for another significant change in SGMLtools. DSSSL (Document Style and Semantics Specification Language) is a language used to specify how SGML documents will look. It helps in translating procedural markup such as “section” to a certain formatting style like “Helvetica Bold, 18 points”, building up tables of contents and more. It is much more powerful than the mapping files used previously, because it can act on context and allows you to define functions. As DSSSL is based on Scheme, you can do just about anything you wish.

We chose to use DSSSL not only because of its power, but also because it is an industry standard (contrary to the old method and to alternatives we evaluated). Also, it helped us jump-start the project because a complete set of DSSSL styles for the DocBook DTD is available.

So, How Does SGMLtools Work?

SGMLtools 2 is a collection of tools based around three core elements:

- the DocBook DTD
- the standard DocBook DSSSL files
- Jade, the SGML/DSSSL parser

When you hand your SGML source to SGMLtools (with the command **sgmltools**), it basically does nothing but call Jade with the name of the SGML file, the name of the DSSSL file to apply to it and the requested output format. The following sections go into some detail in order to make the process clear. It is not difficult to understand, and it helps a great deal when you want to make modifications to have some basic knowledge of what happens during a run of SGMLtools.

Jade first reads the SGML file and tries to find the document type definition from the SGML file's declaration at the beginning of the file. For example:

```
<!DOCTYPE article PUBLIC "-//Davenport//DTD DocBook
V3.0//EN">
```

appears at the beginning of a DocBook-compliant document. (Note that **article** can refer to any part of the DocBook DTD, and **para** can be used to designate a single-paragraph document.) From the **PUBLIC** identifier, Jade obtains the file name of the DTD definition (see the sidebar on Public and System Identifiers), and if all this succeeds, the SGML source is checked for compliance.

After the document has been found to be okay ("validated"), Jade reads the indicated DSSSL file and executes it against the parsed SGML file. The DSSSL "program" reads the SGML document from objects in memory and outputs another memory structure called a **Flow Object Tree** (FOT). The FOT will look structurally like the SGML document, but it contains information on fonts, sizes, and other options. Finally, Jade hands the FOT to one of its backends which converts the generic-style information into the backend's specific file format.

As a short example to illustrate this process, start with an SGML document with the line:

```
<Sect1><Title>Introduction</Title>
...
```

This is a top-level section with "Introduction" as the title. Jade determines it is a valid DocBook document by reading a DSSSL file, perhaps **ldp.dsl** which gives instructions for Linux Documentation Project style formatting.

The following section could be in the DSSSL file:

```
(element SECT1 TITLE ((make paragraph
   font-family-name: "Times New Roman"
         font-weight: 'bold
         font-size: 20pt))
```

This expression says "for TITLE elements within SECT1 elements, output a paragraph with a 20pt bold Times font". Taking some shortcuts, we can say that this expression results in a flow object with the given properties and the text "Introduction" for content (the concept of making a paragraph out of everything, even headings, will be familiar to people who have worked with DTP [distributed transaction processing] software). When everything is done, Jade hands all the flow objects to the backend, for example, TeX. This backend, upon encountering the flow object for our introductory section title, will output something like:

```
{\setfontfam{Times-Roman-Bold}\setfontsize{20pt}Introduction}
```

which can then be processed by TeX and a special TeX package to generate DVI and PostScript.

Note that the beauty of DSSSL is that you talk only about style, not about specific instructions for specific formats. Whether TeX, RTF or groff, you'll always get at least a close equivalent of a “20pt Times New Roman Bold” section header. If you need to tune this, you can easily override pieces of DSSSL specifications for specific backends. Often, you'll at least have different DSSSL files for hardcopy and HTML output.

Customization

One of the biggest advantages of the new version is that it is very easy to customize—once you get the hang of DSSSL. As the previous part showed, you don't even need to know a lot about the backend. In DSSSL, you deal with fairly high-level stuff like font names without worrying about how these font names are dealt with in PostScript or groff documents.

The original DocBook DSSSL style sheets supplied by SGMLtools are meant to be customized. All you need to do is write your own style sheet that includes the original one and overrides what you want to customize, often just a few lines to tune parameters. In SGMLtools you'll find a few examples of these customizations. After you set up your own DSSSL style sheet, you must make sure SGMLtools uses it. Do this by giving the **-d** or **--dsssl-spec** option pointing to your DSSSL style sheet.

Migrating from Linuxdoc

The first question of many Linuxdoc users is, “what about my current documents?” The answer is, you'll have to migrate from Linuxdoc to DocBook within six months from the release date of SGMLtools 2. The package provides a tool to help you in the conversion process.

The first step in the migration process is to make sure your documents are compliant with the latest SGML-Tools 1 version, which will be 1.0.7 or newer. Install this software and run your documents through it to make sure they're up to date.

The second step is to convert your documents with the command **sgmltools --backend=ld2db**, which spits out DocBook documents. If this run succeeds, you can finalize the migration by reading up on DocBook and seeing whether you are satisfied with the result of the conversion. From this point on, you can continue to write in DocBook.

In order to give you some space for planning your conversion, we'll continue to support SGML-Tools 1 for six months after the release date of SGMLtools 2 (which is unknown now, but should occur fairly close to the publication date of this article—check the web site for details). After six months, SGML-Tools 1 will

be removed from the web sites and as far as we are concerned, the Linuxdoc DTD will be history. We'll remind you in comp.os.linux.announce well in advance of this event, and of course, you're free to keep using SGML-Tools 1 for as long as you wish, but we recommend you take the trouble to learn DocBook and start using SGMLtools 2—it'll give you even more flexible formatting power.

Public and System Identifiers

Resources

Cees de Groot has been an avid Linux user since the early days, and he has tried to pay back the great favour Linus did him by contributing small bits and pieces to various parts of the system. Since fall 1996 he has been maintaining SGMLtools. In the daytime, he is a Java consultant working for a small company specializing in Intranets. You can reach him as cg@pobox.com.

Archive Index Issue Table of Contents

Advanced search

Tcl/Tk: The Swiss Army Knife of Web Applications

Bill Schongar

Issue #55, November 1998

Tcl/Tk offers many uses to the web programmer. Mr. Schongar describes a few of them.

While many people think of Tcl/Tk (Tool Command Language, pronounced “tickle”; Tk stands for TCL toolkit) as a great tool for cross-platform GUI (graphical user interface) applications, not as many consider it for web programming tasks. The truth is, its ease of use and flexibility make it a natural choice for CGI (common gateway interface), server-parsed embedded scripting, applications delivered through a plug-in, and even as a tool for creating your own web server from the ground up.

I will present examples of Tcl at work in these situations and also some ideas and additional ways Tcl can be used for whatever web programming task is needed.

CGI Basics in Tcl

Reading environment variables and printing to standard output are the building blocks of CGI. Sure, you want to deal with incoming data, run fun processes and format the output, but first things first. By understanding how any given language performs the basic tasks, you can start building things that actually do something without spending a lot of time mired in details.

Listing 1 is a simple script in Tcl that reads the environment variables set by the web server and sends them back as HTML output to the user. The first thing to note is the choice of shell for interpreting the script. Tcl/Tk distributions include two shells: **wish** and **tclsh**. **wish** is the “windowing” shell, and it carries the overhead of initializing the GUI functions, consuming more than double the system resources of its command-line-only counterpart. For this article, I'm assuming you are running version 8.0 or later of Tcl. (Check by running tclsh and then typing **info tclversion**.)

Next, a standard header is sent back to let the browser know what type of information is coming. In the Listing 1 script, the browser learns the data is HTML text rather than plain text, an image or something else entirely. **puts** is the Tcl equivalent of Perl **print** or C **printf**, complete with **\n** for newlines, and the output can be redirected to a file instead of the default standard out (stdout). So, once the script has told the browser that the input is HTML, it provides the HTML, in this case a title tag and text.

To print out every environment variable, we need to generate a list of which ones exist, then loop through and write out the names and their values. Doing this requires a better sense of how Tcl treats variables and executes commands. First, Tcl doesn't care too much about what type of data is stored in a variable—text or numbers of any length are just fine. Second, variables in Tcl come in three forms: a single variable, an array and a formatted list. To see how these work, look at the following lines of code:

```
set foo "123abc"
set junk(1) "a"
set junk(2) "b"
set bar "a b c d e f g h i j"
```

The **set** function is used to create or modify the values of a variable. In this case, **foo** is a single variable while **junk** is an array and **bar** is a list. What makes **bar** a list is that each character is separated from the other by a space, allowing certain TCL functions to automatically parse the data.

When you see something in square brackets in a Tcl program, it is normally being used to execute what is contained in those brackets, and to substitute the return value for the entire expression. For example, if I have a function called **countdown** which returns the number of seconds remaining until the year 2000, I can easily display that result by writing **puts [countdown]** in my Tcl code. If it returns 300, the number 300 is printed. There are exceptions to this general rule, of course, and we'll run into one of them when we get to parsing user input. So, the line

```
set mylist [array names env]
```

means “create a variable named **mylist** and set it to the result of the command **array names env**”.

Arrays are useful for related groups of information, such as environment variables. The “array” group of functions allows you to search through the names of array elements, as well as many other helpful operations. Executing the command **array names junk** will make a list of all the element names in the array, which right now would be “1 2”. By telling Tcl you want to look through

the **env** array, you obtain a list of names for all the environment variables set by the server, and you store that list in the variable **mylist**.

Looping through each item in **mylist** is made easy by the **foreach** command. It automatically parses a Tcl list and assigns the value of the current element to the variable name you specify. In Listing 1, the name of each environment variable will be stored in **foo** when its time comes, and the **puts** line will show the name of the variable as it originally appeared, followed by its interpreted value. The "$" sign indicates to Tcl that this is a variable and its value should be substituted.

Processing User Data

Since you'll want to do more than just print out some environment variables or static data, here's how to process incoming data in Tcl. The first step involves getting access to the environment variables, which we've covered. This will tell us whether the data is coming in by the **GET** method, stored in the **QUERY_STRING** environment variable, or as a **POST**, stored in standard input. Let's find out using the slightly modified version of our previous program shown in Listing 2. Yes, this program is a lot longer. The extra length, however, comes from dealing with getting the user input, formatting it and storing it in an array. Once I explain how those parts work, I'll show you a much shorter way of writing this program.

The parse (**cgiParse**) and decoding (**urlDecode**) procedures are taken from Brent Welch's *Practical Programming in Tcl/Tk*, with minor modifications. The parse routine is reasonably straightforward. It determines whether the data is stored in **QUERY_STRING** or standard input, then stores it into a variable called **text**.

Special characters can cause problems for CGI programs, so the server encodes percent signs and slashes as their hexadecimal equivalents and spaces as plus signs. Your program must convert the data to its original form. Doing that in C and other languages can be difficult, but Tcl makes it fairly easy. As you can see, each time data is being processed, either for the name or value of the variable, that data is sent to urlDecode. There, the **regsub** command works its magic. To use it, specify (in this order) a search pattern, the original data, its replacement and the variable in which to store it.

Notice that the **foreach** loop at the end of the cgiParse procedure is doing two things that our previous foreach loop didn't do. First, it is specifying more than one temporary value for use in each loop. That is, the first time through the loop, element 1 of the list will be stored in **name**, and element 2 will be stored in **value**. The next loop will use elements 3 and 4, and so on. Any number of variables can be specified in this way, which makes processing long lists a

breeze. The second thing it is doing differently is directly using the results of a command as the list, rather than creating a variable to hold the list first. You can do it either way, but in this example you save a fractional bit of processing time and memory.

Once the parsing and decoding libraries have been laid out, the program starts its run. The content header is sent to the browser, and cgiParse is run in order to store all user-entered values (from a form or some other way) into the variable array **cgi**. Then it loops through each element in the cgi array and prints out the names and values of all the elements.

One benefit to the way the parsing functions are set up is that you can test user-input values on the command line. Since it doesn't rely on finding a GET or POST method, it will get the data wherever possible, defaulting to the command line. So, you could easily test your cgi script before uploading it to the server, without having to create an elaborate wrapper to set environment variables.

Function Libraries—Yours and Everyone Else's

Tcl procedures, or procs, are your subroutines. If you have created some procs, you can easily put them in their own Tcl script, then use the **source** command to load those scripts so they are ready for use. To keep your code to a minimum, you may want to use the cgiParse and urlDecode routines shown in Listing 2. If you saved them as “cgistuff.tcl”, you could rewrite the script in Listing 2 as:

```
#!/usr/bin/tclsh
source cgistuff.tcl
puts "Content-type: text/html \n\n"
cgiParse
foreach foo [array names cgi] {
   puts "Variable: $foo Value: $cgi($foo)"
}
```

The source command loads and executes a Tcl script, so be careful that you don't have any unwanted commands hiding in that script outside of a procedure.

Before you go off writing too many of your own procedures, though, you'll want to take a look at what is already available. A lot of talented people have put time and effort into writing well-documented, very functional procedure libraries, such as Don Libes' cgi.tcl library, which covers everything from basic parsing to cookies and file uploads. (See Resources.)

Data Handling

Sometimes the data you need doesn't come from the user. Product catalogs, maps, schedules and more all come from some sort of external data file, whether real databases like Oracle or Informix or something as simple as a delimited ASCII file. No matter what your data needs are, Tcl can help you out.

Flat files are the easiest to deal with. Open a file, read through line by line until you reach the end or find what you are looking for, then close the file. In Tcl, reading in a file line by line looks like this:

```
set f [open foo.txt r]
while {[gets $f stuff] != -1} {
  # Do something with the line
  # of data
  (`stuff')
}
close $f
```

Just as in Perl or C, you create a file handle from which all subsequent operations work. The **gets** command grabs one line at a time from a file and stores it to a variable. If the return value from gets is 1, you've reached the end of the file. So what do you do with the data once you have it? For the most part, you're going to become fast friends with the **split** and **lindex** commands.

split breaks up a string, either character-by-character or at every occurrence of specified characters and returns a new list of the elements. If you want to access specific elements of the list, lindex allows you to specify the list and the element's position and returns that element's value. Note that elements are numbered starting at 0, so an index value of 1 points to the second element in a list.

A bit higher on the effort scale is processing a special database format, such as a dBASE file or some other defined database format. You may be fortunate enough to find existing filters for this kind of file (two different filters exist for dBASE files), but if you need to write your own, Tcl 8.0 handles binary data quite well. Use the **read** command to grab whatever size byte blocks you want, then use **binary scan** to quickly break up and format it.

If you're concerned about speed or already have a C routine to parse your external data, Tcl makes it easy to create new Tcl commands encapsulated in loadable libraries. For most functions, it's as easy as cutting and pasting into the library framework provided by Tcl and adding some Tcl-specific commands to create or set variables.

When you get to the top of the database world and are dealing with Oracle or Informix, you're already covered. Tcl extensions have been made for Oracle, Informix and probably others by the time you read this article. Most of them

provide access to the SQL layer for the database, but you can also access the lower-level functions of the system. All of them are available on-line, although compiling them sometimes requires access to the commercial libraries shipped with your RDBMS.

Client-Server CGI

One problem with basic CGI is that it doesn't provide for real persistence. Sure, you can use cookies, file-based data on the server side or append horribly long strings to the URL, but none of those is an ideal solution. In addition, if you're loading things like inventory data from a database, you have to account for initialization time and overhead every time the script is run.

In some situations, the best solution is to have a secondary server process running that shares data with Tcl through sockets in a true client-server fashion. In that way, your server could load the needed information and become a persistent data store, for whatever purpose. In Tcl, that's an easier task than you might expect. While the actual code is too lengthy to include in this article, I'll include an overview here and will be happy to provide additional details by e-mail (bills@multimedia.com).

Sockets in Tcl are designed to be easy to use. The **socket** command is used by applications wanting to establish a listening post on a port, as well as clients that want to connect to any server, Tcl or otherwise. How would you listen on a port? Just use:

```
socket -server sayHello 9999
```

Now you have a server listening on port 9999 that will execute the Tcl procedure **sayHello** whenever a new client connects. What if you want an asynchronous socket? Use:

```
socket -server -async sayHello 9999
```

When clients want to connect to you, they just point to your IP address and port using the socket command:

```
socket 10.0.0.1 9999
```

When **sayHello** executes, it receives three arguments: the socket channel your Tcl server has opened to the client, the IP address of the client and the client's port. You can configure the socket channel for the type of buffering and blocking you want, and you'll normally set up a **fileevent** for the channel. A **fileevent** is used to generate notification when the channel becomes either readable or writable (your choice or use both), so that you don't have to poll the socket for new data all the time. Now you and the client are ready to exchange information.

So, once you've decided on what your server will do, your CGI program can parse the data as usual, quickly establish a socket connection, and then let the server process the information.

Extending the Client—Tcl/Tk Plug-in

For some projects, you may want to do more than the browser is able to support. By providing the end user with a plug-in, you get the benefits of being able to run a real application right inside their existing browser without too much of a hassle. One drawback to most plug-ins is that they run only under Microsoft Windows, making them unsuitable for real cross-platform work. Tcl's plug-in doesn't have this problem—you can download precompiled binaries for Linux, Solaris, SunOS and yes, even MS Windows. You may also find that by the time you read this, it has been ported to other platforms as well, such as the Macintosh OS.

Using the plug-in, you can run **Tclets**, which are small Tcl/Tk scripts that run in a restricted (for security reasons) Tcl environment. You and your users can define just how much access you want the plug-in to provide, eliminating or rerouting commands and situations which could be hazardous to your machine's health.

Once you have a Tclet created and your users have the Tcl plug-in, reference it in an HTML page using the

```
<EMBED SRC>
```

tag. So, if your Tclet is called foo.tcl, the tag would look like this:

```
<embed src="foo.tcl" width=400 height=300>
```

If you're wondering what kinds of things have already been made to take advantage of the plug-in, look no further than http://www.tcltk.com/tclets/, which contains everything from Tetris clones to Adaptive Optics demonstrations and VRML editors.

Extending the Server with Tcl—Server-Parsed Tcl and More

Server-parsed HTML has been around for awhile, ranging from basic server-side includes (SSI) to integrated environments complete with database access. It provides dynamically-generated HTML pages without the overhead of calling an external CGI program, and makes it easy even for non-programmers to access all the functionality it provides.

Typically, when a file with a special extension such as .foo is referenced, the server scans through the HTML and looks for special tags. When those tags are found, it executes whatever instructions they contain, then replaces those

sections in the document with the output from the command. Those tags could be anything from the current date to a dynamically generated HTML table with a product price list.

Several solutions exist for using Tcl as a server-parsed scripting language. Two of the most powerful commercial products are NeoWebScript from NeoSoft and Velocigen for Tcl from Binary Evolution. Both products extend the Apache web server with an in-process module, so that they are running all the time in wait mode, ready to do their work. One big difference between the two is that while Velocigen follows the common trend of using a special file extension to identify a file which needs parsing, NeoWebScript follows the more traditional SSI structure of embedding the command in comments. Examples are shown in Listing 3 and Listing 4.

With these more advanced server-side parsers, you can also obtain a level of data persistence through internal variables. For example, you could make a web scavenger hunt on your site to keep a list of the visited pages, and when all the required ones have been seen by a particular user, that user wins. Wins what? I don't know—let marketing worry about it.

Web Servers in Tcl

You won't be able to go out and compete with Apache for market share, but web servers created in Tcl are easy to write, extensible and portable across all platforms. As we saw earlier, sockets are easy to implement in Tcl, which gives you more time to focus on customizing the server to meet your needs, rather than spending it on getting the basics to work.

If you want to see a nice implementation of this concept, take a look at Tcl-HTTPD, freely available from Scriptics. It has CGI support, server-parsed scripting and a host of dynamic configuration options, just to name a few aspects. More basic examples are also available from a variety of Tcl sources on the web, as well as an excellent article by Steve Ball and a white paper by Brent Welch. (See Resources.)

Conclusion

Tcl provides an easy way of addressing almost any web programming issue. With a large development community, a wide selection of extensions and freely available function libraries, it is a web power tool waiting to be discovered. Whether client- or server-side, you get a lot of options without a lot of hassle.

Resources

Bill Schongar can normally be found staring at a screen—writing, playing games or actually doing his job as Senior Developer for LCD Multimedia. If not, he's off with the horses and medieval attire. You can reach him with any questions, comments or random thoughts at bills@lcdmultimedia.com.

Archive Index Issue Table of Contents

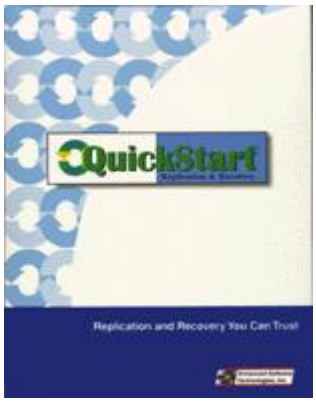
Advanced search

QuickStart: Replication & Recovery 1.2

Daniel Lazenby

Issue #55, November 1998

With QuickStart, bare-metal recovery is reduced to a mere three-step process and a couple of mouse clicks.



- Publisher: Enhanced Software Technologies, Inc.
- E-Mail: info@estinc.com
- URL: http://www.estinc.com/
- Price: $99 US
- Reviewer: Daniel Lazenby

The need for industrial-strength backup and bare-metal recovery tools increases as Linux continues to mature and move into the mission-critical world. Enhanced Software Technologies' (EST) QuickStart product addresses this need.

Total failure of the operating system hard disk might be considered a worst-case system recovery situation. Normally, the replacement disk lacks any operating software or system configuration files. This situation is sometimes referred to as a bare-metal recovery. Typically, a bare-metal system recovery begins with laying down the operating system. Once the operating system has been laid down, a minimal amount of system configuration is done. Next, the

backup software is loaded and configured. With these steps accomplished, the platform can be restored to its baseline configuration. Many hours, CDs, diskettes or tapes later, the operating system and application software will have been restored. QuickStart greatly simplifies this process.

With QuickStart, bare-metal recovery is reduced to a mere three-step process and a couple of mouse clicks. Just pop in the Data Rescue diskette, insert the QuickStart recovery tape and boot the machine. After the machine boots, one is presented with a simple dialog box containing a couple of buttons. Selecting the recover button displays a “Recover System” dialog box. At this point, you verify the source media and target destination. A simple click on the “Start Recovery” button and you are on your way to restoring the system.

Creating Recovery Media

Creating the QuickStart recovery media involves booting the system from the QuickStart diskette. Next, you verify the type of mouse and tape drive and enable compression if you wish. At this point, you can choose to save your configuration or repeat the configuration steps the next time the system boots from the QuickStart diskette.

The QuickStart release I reviewed presented a five-button bar menu once the product was configured. This menu provided the options to back up the system, verify a backup, recover the system, configure QuickStart, or exit and reboot the system. Selecting the backup button displays the “Backup System” dialog box. Here, you select the backup source and the destination media. Selecting the “Start Backup” button on the dialog panel displays the backup progress meter panel. The progress panel displays the lapsed and remaining time, total kilobytes (KB) to be backed up, KB completed and remaining, and the current transfer rate.

Upon completion of the backup, an option to verify the backup is presented. This release of QuickStart required the backup verification to be started manually, and I understand verification will be started automatically with the next release.

General Performance

My backups used an ATAPI-connected Eagle Ti-4 tape drive. Using QuickStart compression, a backup of my 3.14GB disk took less than 50 minutes. The progress display had indicated it would take about 50 minutes to verify the compressed backup. Non-compressed backups took about 1.5 hours. Verification of the uncompressed backup also took about 1.5 hours. I got more reliable verification results on the uncompressed backups than I did on the

compressed backups. My hardware configuration seemed to prefer uncompressed backups—I'm not sure why.

Other Uses for This Product

Besides its usefulness as an integral component of an overall system recovery strategy, QuickStart can be used to move a system's software from a smaller to a larger hard disk. It can also rapidly replicate a standard software installation on multiple platforms. (Contact EST for licensing options and pricing if you plan to do mass system replication.)

Supported Operating Systems

Release v1.2 of QuickStart can be used with several Intel-based operating systems including Linux, UnixWare, SCO Interactive UNIX, BSD, Windows NT, Windows 95, NetWare and OS/2. An upcoming release of QuickStart is expected to support the Power PC-based Macintosh OS, a couple of other flavors of UNIX and the BeOS.

Supported Devices

Devices supported by QuickStart version 1.2 include Seagate, Exabyte and Iomega parallel port TR-1, TR-3 and 2GB tape drives. SCSI adapters supported by QuickStart include Adaptec, Mylex/Buslogic and NCR. The Iomega and SyQuest removable SCSI disk drives are also supported. Floppy tape drives such as the Travan Ditto are supported as well. Older QIC80 tape drives are not supported by Version 1.2 of QuickStart. My Colorado Mountain 250/350 is an example of one such unsupported QIC80 tape drive. I have heard there are plans to support parallel devices such as the Iomega ZIPplus and SyQuest drives. Check the EST web site for an up-to-date list of supported devices.

Manuals and Support

The 23-page user's manual is more of a pamphlet than a manual. It is simply written and easily read. Bold titles and headings make it easy to locate information by “thumbing” through the manual. It also includes backup administrative and management suggestions.

I learned the first source of support should be the vendor who sold you the QuickStart product. In addition to the vendor's support, EST maintains a web site and a support e-mail address. E-mails to the support address received prompt and informative responses. At the time of this writing (August), the web site's support pages were primarily focused on the BRU product. A “Tech Tips” page provided some concepts and tools relating to BRU backups. Hopefully, the EST “Tech Tips” web page will grow to include information on how the BRU and

QuickStart products can be used together to provide reliable bare-metal backup and recovery.

Relationship of QuickStart to BRU

QuickStart and BRU are comparable products. QuickStart provides the means to make an easily restored full system backup. BRU provides the means of making full backups and daily incremental/differential backups. Note that a BRU backup tape cannot be used with the QuickStart diskette. QuickStart uses a sector-by-sector backup to make an image of the disk, and BRU does a file-by-file backup. This difference in backup techniques means QuickStart can read only QuickStart tapes.

Where Can You Buy QuickStart?

QuickStart can be purchased directly from Enhanced Software Technologies, Inc. Additional resellers are listed on Enhanced Software Technologies' web site.

My test system was a Pentium II-based ASUS P2L97 motherboard, with 64MB RAM, 3.2GB Quantum Disk and an IDE-attached Eagle TR-4i Tape Drive.

Daniel Lazenby holds a B.S. in Decision Sciences. He first encountered UNIX in 1983 and discovered Linux in 1994. Today he provides engineering support for a range of platforms running Linux, AIX and HP-UX. He can be reached at dlazenby@ix.netcom.com.

Archive Index Issue Table of Contents

Advanced search

Informix on Linux

Fred Butzen

Issue #55, November 1998

I like the product, and seeing it available for free on my favorite operating system was like having Christmas in July.

- Manufacturer: Informix
- URL: http://www.informix.com/
- Price: Free
- Reviewer: Fred Butzen

On Thursday, July 23, 1998, Linux took a stride closer to the business mainstream. Informix, in response to a groundswell of demand among its users, released its port of Informix to Linux.

For me personally, this was an important development. I have worked with Informix for over 15 years, and first used it on an Altos 586 under Xenix. I like the product, and seeing it available for free on my favorite operating system was like having Christmas in July.

What follows are my first impressions of Informix on Linux.

Informix Release Under Linux

The Informix release consists of the following:

- The Informix Standard Engine is Informix's older technology, which uses the file system to manage data. The Standard Engine is adequate for development work and for managing small- to mid-sized databases. It does not require the resources or management that Informix Online does, but it is not robust enough for enterprise-level work.

- Informix Connect is a set of libraries which manage connections to the database over a network.
- Informix ESQL/C is a kit of libraries, header files, and utilities that let you write C programs that interact with the Informix engine.

To download the package, you must register with Informix first. Informix then e-mails you a serial number and key required for installing the package. The package is free; however, you must still pay for technical support and for runtime licenses to run the bits you develop on customers' machines.

Most significantly, the package does not include ODBC or JDBC drivers. However, Informix CLI includes a Visigenic ODBC driver as part of its development kit. To register to download this development kit, check out http://www.intraware.com/shop/product.html?PLNK=000111. No Linux version is available yet, but kits are now available for Win95, NT and various other flavors of UNIX.

First Impressions

Installation and configuration was straightforward, since I'm familiar with Informix. (See the sidebar “Informix Installation and Configuration” for details.)

Informix offers an SQL engine that is fully compliant with SQL-2 standards. It also offers a stored-procedure language (a rather poor one, in my opinion) and triggers.

Test Environment

I tested Informix on Red Hat 5.0, using the 2.0.29 kernel on a home-brew machine, running a 100 MHz 486 with a 2GB SCSI disk and 32MB of RAM.

A reliable source has tested Informix with Debian 2.0 and its libc5 development package, and reports that Informix works fine on the simple tests to which he subjected it.

Testing

To test Informix, I ran the programs with my **baseball** database that I prepared for the book *The Linux Database*. This set exercises common features of SQL, both in interpreted SQL and in SQL/C.

I found that Informix performs as one expects a major commercial package to perform—without error. I could not get it to crash, return erroneous data or otherwise mess up. On these fairly limited exercises, I found its speed to be more than acceptable.

One problem, however, is that as I exercised Informix, I found that the SQL daemon **sqlexecd** spawned dozens of zombie processes. Clearly, work remains to be done.

Pluses and Minuses

Informix on Linux offers a number of pluses to anyone who develops database applications under Linux:

- The main plus, of course, is name recognition: It is well-known and has credibility among potential customers that other packages, such as Yard or mysql, do not. This may not be fair, but it is nonetheless true.
- Informix is a full, robust implementation of SQL-2. Even the best freeware packages have limitations in their implementation of SQL that require workarounds.

However, Informix has some serious drawbacks as well:

- The package offered is Informix's trailing-edge technology. It does not offer the Online engine, data blades or Informix's other enterprise-scale tools. Informix's standard-engine product is a good, solid package, but it is not where the action is.
- The only interface language offered so far is C. If you wish to build web interfaces with Perl, this package will not help you (although there may be a third-party package available that includes a Perl interface to Informix).
- Documentation is particularly helpful in getting Informix up and running. However, it is hard to come by—you must either purchase documentation, which is expensive, or download and print it, which is also expensive.
- Your customers still have to purchase a runtime license for Informix, which can be quite expensive. When you add in the cost of connectivity tools (ODBC, et al.), a customer can easily spend thousands of dollars per installation.

In conclusion, my first impression is that Informix on Linux is a solid, useful package. It is not clear just how many opportunities it will open up for Linux database developers. However, Informix's port to Linux raises Linux's reputation among people who have power—always a reason to celebrate.

Acknowledgment

I wish to thank Jeff Noxon, who shared his impressions of the Informix release with me and who pointed out the URL at which the Informix CLI kit is available.

Installation and Configuration

Fred Butzen is a technical writer and programmer who lives in Chicago. He is the principal author of the manual for the Coherent operating system, and is co-author of The Linux Database (MIS:Press, 1997) and *The Linux Network* (MIS:Press, 1998). He can be contacted at fred@lepanto.com.

Archive Index Issue Table of Contents

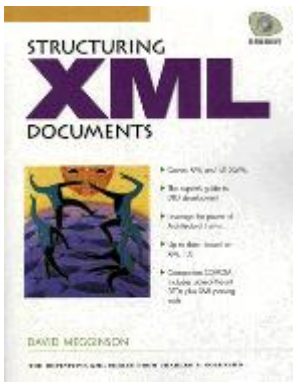
Advanced search

Structuring XML Documents

Terry Dawson

Issue #55, November 1998

XML has nearly all of the power and features of SGML, but will probably be much better supported because of the web-driven market for browsers and editors.



- Author: David Megginson
- Publisher: Prentice Hall
- E-mail: info@prenhall.com
- URL: http://www.prenhall.com/
- Price: $44.95 US
- ISBN: 0-13-642299-3
- Reviewer: Terry Dawson

Take a close look at any of the various documentation projects operating within the Linux community, and you will find SGML. The Linux Documentation Project, the Debian Documentation Project and others are using SGML as the primary tool in producing consistently structured and styled documentation. The search for a more sophisticated replacement for HTML has led to the development of XML, which is based heavily on SGML. XML has nearly all of the power and features of SGML, but will probably be much better supported

because of the web-driven market for browsers and editors. For this reason, XML will probably replace SGML in many applications.

SGML and XML both provide a means of describing the structure of a document. SGML and XML rely on definitions called DTDs, Document Type Definitions, that describe document structure.

In this book, David Megginson competently explains the process of good quality DTD design. While the title suggests it is XML-specific, it is not. SGML and XML have so many similarities that it is possible to describe both simultaneously, highlighting differences between the two where they arise. This book is another in the Charles F. Goldfarb series, and in it, Mr. Megginson describes document structuring using both SGML and XML, managing to avoid confusing the reader during the process. The book has four main parts, and includes a CD-ROM with software that implements XML parsers, and a selection of modern and popular DTDs.

Part One provides some background on XML, describes how it is different from SGML and examines five popular and useful DTDs. This chapter isn't for people with no prior SGML or XML experience and isn't designed to teach you either, but if you are familiar with at least one of them, it will assist you in learning about the other. The chapter on DTD syntax clearly illustrates the differences and similarities between the two. DTDs examined in detail are:

- ISO-12083
- DocBook
- Text-Encoding Initiative (TEI)
- MIL-STD-38784 (CALS)
- HyperText Markup Language (HTML 4.0)

The first four of these are in common use and have inspired many other DTD designs. The CALS table design, for example, has been borrowed many times and used in other DTDs.

Part Two covers the principles of DTD analysis. The core material of the book begins in these chapters. They describe how to critically analyse a DTD from three important perspectives: ease of learning, ease of use and ease of processing. The ease with which a particular DTD can be learned is critically important in having a DTD accepted by authors. If the DTD is difficult to learn, authors will tend not to use it, use only a small subset of it, or worse, misuse it by bending it to suit their needs. Mr. Megginson describes how to analyse the ease of learning of a DTD with the aim of instructing you how to design easy-to-learn DTDs.

The chapter entitled “Ease of Use” describes how to analyse a DTD to determine if it will be easy for authors to use when they are writing their documentation. Some of the issues explored are the naming of tags and attributes, when to use a new tag and when to add an attribute to an existing tag, and structural issues that can simplify or complicate an author's job.

The chapter on ease of processing is of particular interest to those who publish and develop processing tools. A DTD may be easy for the author to learn and use, but this doesn't always translate into something that is easy to process into printed or published form. The lessons are mostly common sense applied to the specific task of DTD design.

The third part of the book covers a number of advanced DTD maintenance and design issues. It will be of interest mostly to people who intend to use SGML or XML for purposes other than publishing, such as database systems or other information management applications. The first topic covered is that of DTD compatibility. I mentioned earlier that the CALS table design had been borrowed for use in other DTDs. When DTDs are similar, it is fairly simple to translate a document from one DTD to another. This is very useful if you wish to exchange documentation with a group which has a different DTD. This chapter describes how to identify compatibility and the advantages of keeping compatibility in mind when designing a DTD.

The second topic extends this discussion to exchanging document fragments. A document fragment might be a single chapter or paragraph from a book. If you wish to share portions of a document with a group using a different DTD, you will find useful tips in this chapter on ways to simplify the task.

The final topic in this section is DTD customisation. DTD customisation is the process of taking an existing DTD and modifying it to suit your specific purposes. Designing a sophisticated DTD can be a complex task. Often, there is little reason to design a DTD from scratch; an existing DTD may provide 95% of what you need, requiring only a small amount of customisation to fully meet your needs. This can save a lot of time and provides advantages in terms of document exchange and compatibility. This chapter describes how to customise DTDs, and how to design DTDs that are easy to customise. The DocBook DTD, for example, was designed with hooks in place that allow for easy customisation.

The fourth and final part of the book covers DTD design using a technique called Architectural Forms. Architectural Forms allow DTD designers to specify the method by which their DTD should be translated into one or more other DTDs. Architectural Forms allow you to write documents which are simultaneously valid for a number of different DTDs. This section of the book

describes the concepts and the implementation of Architectural Forms and offers useful hints and advice to designers wishing to use this facility. I found this part of the book a little difficult to comprehend, but that was almost certainly due to my limited exposure to applications requiring use of this advanced technique. I'm confident that anyone with an application for Architectural Forms will find the information presented to be a good introduction to the topic.

I am pleased to report that the CD-ROM includes Linux versions of the XML parsing software. Two XML parsers are provided. The first is a precompiled version of the popular “SP” parser. The second is a Java-based XML parser called “Aelfred”. Each DTD described in the book is included in its SGML form, as well as a number of links to useful resources on the Internet. The CD-ROM provides some HTML-based documentation, but is otherwise not well documented. I am left with the impression that the CD-ROM was a last-minute addition to the book; nevertheless, it does provide tools to allow the reader to experiment with the techniques described, and to that end it is adequate.

I found *Structuring XML Documents* to be an interesting and informative book that I will certainly be using as a reference in the future. David Megginson has done a nice job of concisely capturing a lot of material while keeping the pace slow enough to allow one to absorb the information fairly comfortably. The book is ideal for both SGML or XML designers, and SGML designers should not be misled by the title. I recommend that anyone with an interest in DTD design, especially those involved with Linux-related documentation projects, take a look at this book. It is certain to be of assistance in your efforts.

Terry Dawson is a full-time Linux advocate who is reveling in his new role of “Dad” with the birth of his son Jack. Mr. Dawson can be reached at terry@perf.no.itg.telecom.com.au.

Archive Index Issue Table of Contents

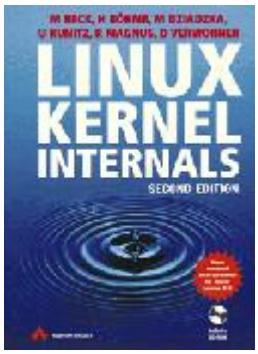
Advanced search

Linux Kernel Internals, Second Edition

Karl Majer

Issue #55, November 1998

It covers each aspect of the kernel in great depth and at a very high level of detail.



- Authors: Michael Beck, Harold Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner
- Publisher: Addison-Wesley
- E-mail: info@awl.com
- URL: http://www.awl.com/
- Price: $45 US
- ISBN: 0-201-33143-8
- Reviewer: Karl Majer

Linux Kernel Internals is an exceptionally well-written book on the internal workings of the 2.0 kernel. It covers each aspect of the kernel in great depth and at a very high level of detail.

The book starts with an introduction to Linux and then ushers the reader along to the methodology of building a kernel. The reader is then introduced to the primitives of the kernel itself, its data structures, algorithms and system calls. At this point the user should have a reasonable understanding of the basics of the kernel.

The author's discussion then migrates to memory management within the Linux kernel. An introduction on architecture-independent memory management begins this section. Other covered points of interest concerning Linux memory management are the assignment of virtual address space for the proc file system, the methodology chosen by the kernel to do caching of block devices, and the manner in which the Linux kernel pages.

Interprocess communication is the next topic. The reader learns the methods by which the kernel synchronizes communication among its various parts. A brief discussion is held concerning communications via files, sockets and pipes.

Next, the Linux file system is introduced. The basics of a file system are described, as well as how the kernel sees the file system. Following that is the Linux implementation of the proc file system, and finally an excursion into the workings of the EXT2 file system.

An operating system wouldn't be complete without device drivers to allow the end user to attach and use any additional devices he may have. The book covers character devices and polling devices in-depth, and then moves on to explain the difference between writing polling device drivers versus interrupt-driven device drivers. The user is then led through a tutorial on implementing a device driver to manipulate the PC speaker.

One of the true delights of the Linux operating system is its inherent and native network support. Unlike other non-UNIX operating systems, Linux is quite capable of using network services without the installation of special software. The Linux kernel handles communication at a very low level, and the book discusses this and the kernel's handling of network devices with a great deal of depth. The various communication protocols, i.e., IP, TCP, UDP, ARP, and the manner in which they are implemented are also discussed.

In order for the Linux kernel to support a device, protocol or even another processor, the kernel must “know” of these things by including the source code required to support the items internally. While this is not a particularly difficult thing to do, it does however increase the size of the kernel which runs in memory. A method by which the kernel can load only the necessary objects needed for operation, otherwise known as modules, was developed.

An introduction to the modules in the kernel can be obtained from the excellent section on modules in this book. The discourse covers items such as the types of kernel objects which can be made into modules, and the kernel daemon, the program which runs in memory and handles the dynamic usage of modules. Finally, there is an elegant tutorial on the actual writing and debugging of a module.

Given the availability and low cost of components today, people naturally desire multiple processors in their computers. Although disappointingly brief, a section on SMP (Symmetrical Multi-Processing) is included in the book. While this section does contain a good deal of information, it is comparatively sparse and Intel-centric.

At the conclusion of the primary discussion, the book offers four appendices which contain a wealth of information. Each system call available to the Linux programmer is discussed in detail, as are all of the commands related to obtaining kernel information. An in-depth look at the proc file system and information about the running kernel which it can provide is examined. Finally, the reader is given a fairly good look at the boot processes and all that is involved in starting the Linux system.

All in all, *Linux Kernel Internals* is a very good book for high level, in-depth reading about the kernel. The only drawback would be its Intel-based twist, which while acceptable, is not exemplary of the strongest feature of Linux—its ability to run on many hardware platforms both old and modern such as DEC Alpha, Sun SPARCs, Macintosh and SGI.

Karl Majer is currently a UNIX system administrator for America Online. In his little spare time, he enjoys spending time with his family, programming, reading and promoting Linux. Feel free to send e-mail to him at majer@bitbucket.org.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Beginner's Guide to JDK

Gordon Chamberlin

Issue #55, November 1998

This article covers the use of the Java Development Kit on a Linux platform. It includes a general introduction to Java, installing the JDK 1.1.6, compiling Java support into the Linux kernel, writing a simple Java program and studying an example.

Although several ongoing projects have the goal of porting Sun's Java Development Kit (JDK) to Linux, for the purpose of this article I will look at the largest and most stable effort. It is hosted by the Blackdown Organization, with much of the effort coming from Randall Chapman (for JDK 1.0) and Steve Byrne (for JDK 1.1). At the time of this writing (August 1998), JDK 1.1.6 version 3 was in beta testing. This version will be publicly released by the time this magazine is printed. See the Blackdown's web page at http://www.blackdown.org/java-linux.html for up-to-date information on the JDK port for Linux.

Note that I use the Caldera Open Linux 1.2 Standard distribution of Linux, so I will discuss the JDK installation with respect to that distribution. The JDK 1.1.6 port to Linux is quite stable on Caldera Open Linux 1.1 and 1.2 out-of-the-box installations. It will also work with many other distributions of Linux.

The Blackdown Java Linux porting effort still needs volunteers. Please feel free to contact them. In fact, I received an e-mail message from Steve Byrne stating that the source code is not hard to understand and many areas besides debugging need volunteers. He also said many areas of specialization exist in the source code, so if you are good at one thing but not another, please don't let that stop you from volunteering.

I will review the high points of three different JDK files, drawing on personal experience. The main source of information is the Java Linux FAQ at the Blackdown site. The first file is http://www.blackdown.org/java-linux/docs/faq/FAQ-java-linux.html. The two other files are the README that comes with Sun's JDK and the README.linux that comes with the Linux port of the JDK. These two

files are found in the jdk1.1.6 directory after installing the JDK. There is also a link to README.linux from the Java-Linux FAQ.

A Short Introduction to Java

Two types of Java programs exist: applications and applets. Applets are simply small Java programs that run within the context of a web browser. Applications are stand-alone programs.

Both applications and applets start as Java source code files. When a Java program is compiled, the source code is turned into Java byte code. In general, a byte-code file is generated for each “class” declared in your source-code file. These byte-code files have the extension “.class”.

The byte code is then interpreted by the Java Virtual Machine (JVM). Unless you have a machine that implements Java byte code in hardware, the JVM is a program run by the operating system you are using. This is the case under Linux.

As I stated before, the difference between applications and applets is revealed by the way each is started. An application is started from the command line. It is a stand-alone program. In contrast, an applet is started by a web browser.

When an application is written, a method called **main** is defined. Execution of the application starts in main. To start an application at the command line, type **java *`SomeApplication`***. Please note that you do not type in the .class extension.

An applet has a method called **init**. In addition to the applet byte code, an HTML file exists which contains an <APPLET ...> </APPLET> tag pair defining the location of the byte code and other useful information. When the applet is started from within a web browser, init is called to start the applet. An applet can be started from the command line with a program called **appletviewer**. This program, distributed with the JDK, takes the name of an HTML file, finds all applet tags and runs those applets.

Both main and init can be implemented in a single Java program. The resulting program can therefore be started by either **java *`SomeApplication`*** or with a web browser.

Installing the JDK

As I mentioned above, the Blackdown Organization is the repository for the largest Java Linux effort. This site contains distribution locations, e-mail lists and

known problems with the current JDK for Linux. Currently, the latest release of the Java Development Kit for Linux is JDK 1.1.6 version 3.

JDK comes in two flavors: one for libc and one for glibc. For an explanation of the differences, see the Java-Linux FAQ. According to the README.linux file that comes with the JDK, a good way to determine which version you need is by looking at the libraries installed on your system. This can be done with the following command:

```
ls -l /lib/libc.so.*
```

If the files are libc5, you should download the libc version. If they are libc6, then you should download the glibc version.

To download the JDK distribution, point your web browser to http://www.blackdown.org/java-linux/mirrors.cgi. This page lists sites that distribute the JDK. You can obtain the Linux JDK port only from a mirror of Blackdown.

Take a look at the mirrors, and choose a download site near you. Since Caldera uses libc5, I followed the links to JDK-1.1.6/i386/libc/v3/. Download the file jdk1.1.6-v3-libc.tar.gz. You can also download other files from this directory.

Choose a directory to unpack the distribution using the tar command. I chose /usr/local/. If you choose a different directory, use that directory in place of /usr/local/ wherever it appears in the rest of this article. Once you've picked a directory, go to it using **cd**, then type:

```
tar xzf jdk.1.1.6-v3-libc.tar.gz
```

The installation of JDK is complete. You should now visit JavaSoft to download documentation. Please see http://www.javasoft.com/docs/index.html for download and installation instructions.

Environment Variable Settings

To complete the setup, you should modify your PATH environment variable to include the location of JDK wrappers. Using your favorite editor, edit the appropriate startup file (for me, this was .profile) and add /usr/local/jdk1.1.6/bin to your PATH. Adding this to the beginning of your current PATH setting ensures that this JDK is invoked.

You also need to add two new environment variables: **JAVA_HOME** and **CLASSPATH**.

JAVA_HOME tells JDK where its base directory is located. Although it isn't mandatory to set this variable since the JDK does a good job of determining this location, it is used by other Java programs such as the Swing Set.

CLASSPATH can be confusing and frustrating, but it is possible to use it well and correctly from the beginning. Just remember this simple analogy: **CLASSPATH** is for Java what PATH is for a shell on your machine. Looking closer at the analogy, your shell executes only those programs or scripts residing in the directory pointed to by PATH, unless the full path of the program is specified. **CLASSPATH** works the same way for Java. Only those applications and applets in the directories specified by the **CLASSPATH** environment variable can be run without specifying the complete location.

I usually set **CLASSPATH** to a simple dot. This lets me run any application that is in my current directory. I also create scripts that set my **CLASSPATH** on an "as needed" basis, depending on what I am doing during that particular session.

If you use bash as your shell, these three environment variables can be set as follows:

```
PATH=/usr/local/jdk1.1.6/bin:$PATH
CLASSPATH=.
JAVA_HOME=/usr/local/jdk1.1.3"
export PATH CLASSPATH JAVA_HOME
```

Note that the RPMs which come with Red Hat 4.1 and 4.2 do *not* work out of the box. I recommend erasing the RPMs and using the JDK distribution from Blackdown. Erase the RPMs with the commands **rpm -e jdk** and **rpm -e kaffe**.

Testing the JDK Installation

You're now ready to test the JDK. Either log in or execute the startup file to set your new environment variables, and make sure the new environment variables are indeed taking effect. Executing **rlogin *localhost*** will do the trick.

Now, type **java**. A message giving usage parameters should appear. Typing **javac** should also work, displaying different usage parameters.

Next, use your favorite editor and type in your first Java program; name this file HelloLinux.java.

```
public class HelloLinux {
   public static void main (String args[]) {
      System.out.println("Hello Linux!");
      }
   }
```

To compile this program, type **javac HelloLinux.java**. The compilation process creates a single file called HelloLinux.class. To run your Java application, enter **java HelloLinux**. This outputs the single line “Hello Linux!”

Compiling Support for Java Byte Code into the Kernel

The Linux kernel is capable of detecting Java byte code and automatically starting Java to run it. This eliminates the need to type **java** first. When the kernel is configured with Java support, you need do only two things. First, change permissions of your .class file to make it executable using the **chmod** command. Then, run it like any normal script or executable program.

For example, after compiling the Java program HelloLinux, perform the following commands:

```
chmod 755 HelloLinux.class
./HelloLinux.class
```

Note that you now have to specify the full name of the application. This includes the .class extension.

To set up Java support, you need the source code to the Linux kernel. The default installation of Caldera OpenLinux installs the kernel source code for you. Use this or download the latest and greatest kernel source and install it.

If you haven't compiled a kernel for your Linux box before, I recommend doing it once or twice to get a feel for it. This will also ensure that problems unrelated to Java don't arise when you are trying to add native Java support to the kernel.

Three steps are required to set up the kernel to automatically run Java byte code. You can find more information about using this feature of the kernel in Documentation/java.txt in your kernel source tree.

1. In the “Code Maturity Option” menu, select “Prompt for development and/or incomplete code/drivers”. The support of Java is still somewhat new and may have problems which not everyone is prepared to encounter.
2. In the “General Setup” menu, select “Kernel Support for Java Binaries”. Mark it as either a module or a part of the kernel.
3. Before compiling the kernel, edit the fs/binfmt_java.c file and place the path to your java interpreter in the **#defines** located at the start of that file. (For me, this path is /usr/local/jdk1.1.6/bin/java/.) Also, edit the path pointing to the applet viewer. An alternate method is to leave the paths alone in fs/binfmt_java.c and make symbolic links to the appropriate locations.

If you compiled Java support as a part of the kernel—i.e., it was *not* a module—then there is still another way to tell the kernel where your java wrapper lives. Log in as root and issue the command:

```
echo "/path/to/java/interpreter" >\
/proc/sys/kernel/java-interpreter
```

Note that this command needs to be executed each time you boot the kernel, so you should place it in the rc.local file or an equivalent location.

Running the Demos

JDK includes many examples in addition to the applets and applications. Change the current path to /usr/local/jdk1.1.3/demo/ and you will see the numerous directories containing examples in this directory. Some of the examples include Tic-Tac-Toe, a graphics test and a molecule viewer.

To run Tic-Tac-Toe, use your web browser to open the file /usr/local/jdk1.1.6/demo/TicTacToe/example1.html.

The graphics test is located in jdk1.1.6/demo/GraphicsTest/. It is noteworthy for having been written to be executed as both an application and an applet.

The molecule viewer is an applet located in jdk1.1.6/demo/MoleculeViewer/. I mention it just because it is cool—it shows off the power of Java.

The Possibilities: Real Time Linux Statistics

The company I work for, Visualize, Inc. (http://www.visualizetech.com/), specializes in high-end data-graphing software also known as data visualization. Our products all derive from our core library called VantagePoint. VantagePoint is 100% Pure Java certified. Writing our products in Java allows us to develop the software once, then run it on any platform for which Java has been ported. In practice, our customers have encountered few problems with the products as a cross-platform software package. In fact, the earliest version of VantagePoint was developed completely on Linux. Linux still plays an important role in our company.

VantagePoint products, as a graphing solution, view the world as **data sets**. The possible operations on a data set are **graphing**, **loading** and **analyzing**. After an analysis has been performed on a data set, it is possible to do any of these operations again. A useful sequence of operations is opening and loading a data set, graphing the data to get a visual representation, analyzing the data and then graphing the result of the analysis.

One VantagePoint product, DP Server, is a manager for any data sets you may have. It also allows connections from applets running in a web browser on either an Intranet or the Internet.

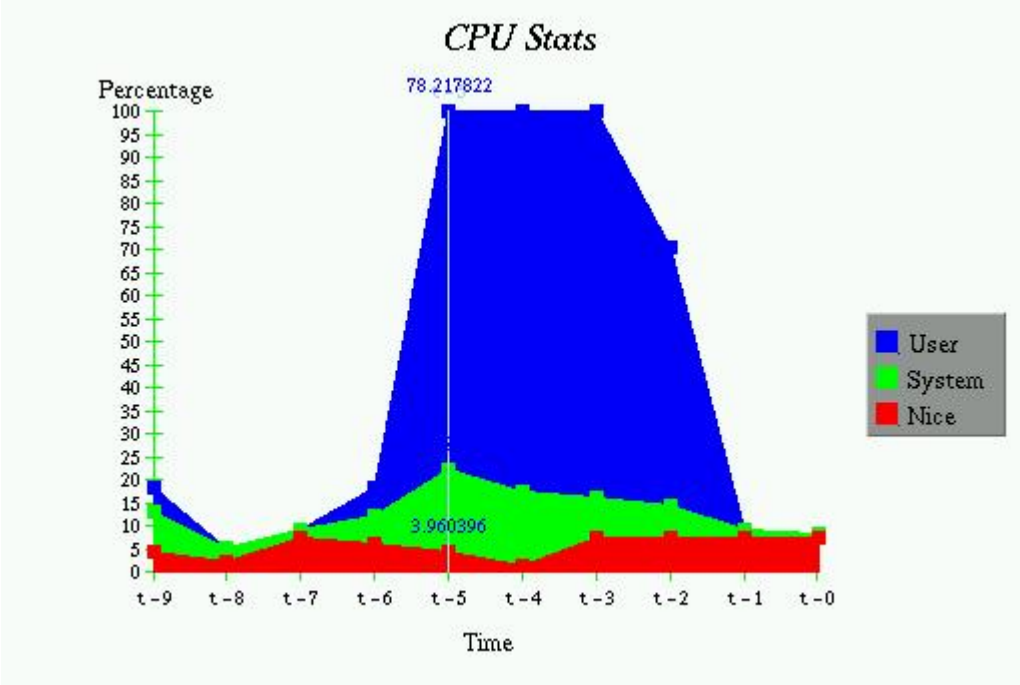


Figure 1: Real-Time Linux CPU Usage Statistics

Now, go to the page at http://www.visualizetech.com/lj/. Follow the link to "Linux Statistics". This page, which looks like Figure 1, starts a Java applet that connects to a DataPoint Server to show a two-dimensional line chart. It is updated regularly with each update showing the current CPU usage on the Linux web server. During this particular snapshot, the Linux box had just finished compiling the HelloLinux.java program.

Conclusion

Java is a boon to the software development industry. Java and Linux offer the best combination the computer industry has to offer: a free, dependable operating system and a platform-independent software language.

Acknowledgements

Gordon Chamberlin is a programmer and system administrator for Visualize, Inc., in Phoenix, AZ. He was introduced to Linux at 1.2.x and quickly bought a 486 to be able to use Linux at home. He enjoys playing computer games, especially Descent II and spending time with his wife, Barbara. Please send comments, questions or jokes to glac@visualizetech.com.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

init

Alessandro Rubini

Issue #55, November 1998

init is the driving force that keeps our Linux box alive and is also the one that can put it to death. This article summarizes why init is so powerful and how you can instruct it to behave differently from its default behaviour. (Yes, init is powerful, but the superuser rules over init.)

In UNIX parlance, the word “init” doesn't identify a specific program, but rather a class of programs. The name “init” is used generically to call the first process executed at system boot—actually, *the only* process that is executed at system boot. When the kernel is finished setting up the computer's hardware, it invokes init and gives up controlling the computer. From that point on, the kernel processes only system calls without taking any decisional role in system operation. After the kernel mounts the root file system, everything is controlled by init.

Currently, several choices of init are available. You can use the now-classic program that comes with the SysVinit package by Miquel van Smoorenburg, **simpleinit** by Peter Orbaek (found in the source package of util-linux), or a simple shell script (such as the one shown in this article, which has a lot less functionality than any C-language implementation). If you set up embedded systems, you can even run the target application as if it were init. Masochistic people who dislike multitasking could even port **command.com** to Linux and run it as the init process, although you won't ever be able to restrict yourself to 640KB when running a Linux kernel.

No matter which program you choose, it needs to be accessed with a path name of /sbin/init, /etc/init or /bin/init, because these path names are compiled in the kernel. If none of them can be executed, then the system is severely broken, and the kernel will spawn a root shell to allow interactive recovery (i.e., /bin/sh is used as the init process).

To achieve maximum flexibility, kernel developers offer a way to select a different path name for the init process. The kernel accepts a command line option of **init=** exactly for that purpose. Kernel options can be passed interactively at boot time, or you can use the **append=** directive in /etc/lilo.conf. Silo, Milo, Loadlin and other loaders allow specifying kernel options as well.

As you may imagine, the easiest way to get root access to a Linux box is by typing **init=/bin/sh** at the LILO prompt. Note that this is not a security hole *per se*, because the real security hole is physical access to the console. If you are concerned about the **init=** option, LILO can prevent interaction using its own password protection.

The Task of init

Now we know that **init** is a generic naming, and almost anything can be used as init. The question is, what is a real init supposed to do?

Being the first (and only) process spawned by the kernel, the task of init consists of spawning every other process in the system, including the various daemons used in system operation as well as any login session on the text console.

init is also expected to restart some of its child processes as soon as they exit. This typically applies to login sessions running on the text consoles. As soon as you log out, the system should run another **getty** to allow starting another session.

init should also collect dead processes and dispose of them. In the UNIX abstraction of processes, a process can't be removed from the system table unless its death is reported to its parent (or another ancestor in case its parent doesn't exist anymore). Whenever a process dies by calling *exit* or otherwise, it remains in the state of a zombie process until someone collects it. **init**, being the ancestor of any other process, is expected to collect the exit status of any orphaned zombie process. Note that every well-written program should reap its own children—zombies exist only when some program is misbehaving. If init didn't collect zombies, lazy programmers could easily consume system resources and hang the system by filling the process table.

The last task of init is handling system shutdown. The init program must stop any process and unmount all the file systems when the superuser indicates that shutdown time has arrived. The **shutdown** executable doesn't do anything, it only tells init that everything is over.

As we have seen, the task of init is not too difficult to implement, and a shell script could perform most of the required tasks. Note that every decent shell collects its dead children, so this is not a problem with shell scripts.

What *real* init implementations add to the simple shell script approach is a greater control over system activity, and thus a huge benefit in overall flexibility.

Using /bin/sh as a Minimal Choice

As suggested above, the shell can be used as an init program. Using a bare shell (**init=/bin/sh**) simply causes a root shell to open in a completely unconfigured system. This section shows how a shell script can perform all of the tasks you need to have in a minimal running system. This kind of tiny init can be used in embedded systems or similar reduced environments, where you must squeeze every single byte out of the system. The most radical approach to embedded systems is directly running the target application as the init process; this results in a closed system (no way for the administrator to interact, should problems arise), but it sometimes suits the setup. The typical example of a non-init-driven Linux system is the installation environment of most modern distributions, where /sbin/init is a symbolic link to the installation program.

Listing 1 shows a script that can perform acceptably as init. The script is short and incomplete; in particular, note that it runs only one getty, which isn't restarted when it terminates. Be careful if you try to use this script, as each Linux distribution chooses its own flavour of getty. Type **grep getty /etc/inittab** to know what you have and how to call it.

The script has another problem: it doesn't deal with system shutdown. Adding shutdown support, however, is fairly easy; just bring everything down after the interactive shell terminates. Adding the text shown in Listing 2 does the trick.

Whenever you boot with a plain **init=/bin/sh**, you should at least remount the root file system before you do anything; you should also remember to do **umount -a** before pressing **ctrl-alt-del**, because the shell doesn't intercept the three-finger salute.

simpleinit, from util-linux

The util-linux package includes a C version of an init program. It has more features than the shell script and can work well on most personal systems, although it doesn't offer the huge amount of configurability offered by the SysVinit package, which is the default on modern distributions.

The role of simpleinit (which should be called init to work properly) is very similar to the shell script just shown, with the added capability of managing single-user mode and iterative invocation of console sessions. It also correctly processes shutdown requests.

simpleinit is interesting to look at, and well-documented too, so you might enjoy reading the documentation. I suggest using the source distribution of util-linux to get up-to-date information.

The implementation of simpleinit truly is simple, as its name suggests. The program executes a shell script (/etc/rc) and parses a configuration file to determine which processes need to be respawned. The configuration file is called /etc/inittab, just like the one used by the full-featured init; note, however, that its format is different.

If you plan to install simpleinit on your system (which most likely already includes SysVinit), you must proceed with great care and be prepared to reboot with a kernel argument of “init=/bin/sh” to recover from unstable situations.

The Real Thing: SysVinit

Most Linux distributions come with the version of init written by Miquel van Smoorenburg; this version is similar to the approach taken by System V UNIX.

The main idea is that the user of a computer system may wish to operate his box in one of several different ways (not just single-user and multi-user). Although this feature is not usually exploited, it is not so crazy as you might imagine. When the computer is shared by two or more people in one family, different setups may be needed; a network server and a stand-alone playstation can happily coexist in the same computer at different runlevels. Although I'm the only user of my laptop, I sometimes want a network server (through PLIP) and sometimes a netless environment to save resources when I'm working on the train.

Each operating mode is called a “runlevel”, and you can choose the runlevel to use at either boot or runtime. The main configuration file for init is called /etc/inittab, which defines what to do at boot, when entering a runlevel or when switching from one runlevel to another. It also tells how to handle the three-finger salute and how to deal with power failure, although you'll need a power daemon and a UPS to benefit from this feature.

The inittab file is organized by lines, where each line is made up of several colon-separated fields: **id:runlevel:action:command**.

The **inittab(5)** man page is well written and comprehensive as a man page should be, and I feel it is worth repeating one of its examples—a stripped-down /etc/inittab that implements the same features and misfeatures of the shell script shown above:

```
id:1:initdefault:
rc::bootwait:/etc/rc
1:1:respawn:/sbin/getty 9600 tty1
```

This simple inittab tells init that the default runlevel is “1”, at system boot it must execute /etc/rc, and when in runlevel 1 it must respawn forever the command **/sbin/getty 9600 tty1**. You're not expected to test this script out, because it doesn't handle the shutdown procedure.

Before proceeding further, however, I must fill in a couple of gaps. Let's answer two common questions:

- “How can I boot into a different runlevel than the default?” Add the runlevel on the kernel command line; for example, type **Linux 2** at the LILO prompt, if “Linux” is the name of your kernel.
- “How can I switch from one runlevel to another?” As root, type **telinit 5** to tell the init process to switch to runlevel 5. Different numbers indicate different runlevels.

Configuring init

Naturally, the typical /etc/inittab file has many more features than the three-line script shown above. Although **bootwait** and **respawn** are the most important actions, several other actions exist in order to deal with issues related to system management. I won't discuss them here.

Note that SysVinit can deal with **ctrl-alt-del**, whereas the versions of init shown earlier didn't catch the three-finger salute (i.e., the machine would reboot if you pressed the key sequence). Those interested in how this is done can check *sys_reboot* in /usr/src/linux/kernel/sys.c. (If you look in the code, you'll note the use of a magic number 672274793: can you imagine why Linus chose this number? I think I know the answer, but you'll enjoy discovering it yourself.)

Let's look at how a fairly complete /etc/inittab can take care of everything required to handle the needs of a system's lifetime, including different runlevels. Although the magic of the game is always on display in /etc/inittab, several different approaches to system configuration can be taken, the simplest being the three-line inittab shown above. In my opinion, two approaches are worth discussing in some detail; I'll call them “the Slackware way” and “the Debian way” from two renowned Linux distributions that chose to follow them.

The Slackware Way

Although it has been quite some time since I last installed Slackware, the documentation included in SysVinit-2.74 tells me that it still works the same. It has fewer features but is much faster than the Debian way. My personal 486 box runs a Slackware-like /etc/inittab just for the speed benefit.

The core of an /etc/inittab as used by a Slackware system is shown in Listing 3. Note that the runlevels 0, 1 and 6 have a predefined meaning. This is hardwired into the init command, or better, into the shutdown command part of the same package. Whenever you want to halt or reboot the system, init is told to switch to runlevel 0 or 6, thus executing /etc/rc.d/rc.0 or /etc/rc.d/rc.6.

This works flawlessly because whenever init switches to a different runlevel, it stops respawning any task not defined for the new runlevel; actually, it kills the running copy of the task. In this case, the active task is /sbin/agetty.

Configuring this setup is fairly simple, as the roles of the different files are clear:

- /etc/rc.d/rc.S is run at system boot, independently of the runlevel. Add to this file anything you want to execute right at the start.
- /etc/rc.d/rc.M is run after rc.S is over, only when the system is going to runlevels 2 through 5. If you boot at runlevel 1 (single user), this script is not executed. Add to this file anything you run only in multiuser mode.
- /etc/rc.d/rc.K deals with killing processes when going from multi-user to single-user mode. If you add anything to rc.M, you'll probably want to stop it from rc.K.
- /etc/rc.d/rc.0 and /etc/rc.d/rc.6 shut down and reboot the computer, respectively.
- /etc/rc.d/rc.4 is executed only when runlevel 4 is entered. This file runs the “xdm” process, to allow graphic login. Note that no getty is run on /dev/tty1 when in runlevel 4 (this can be changed if you wish).

This kind of setup is easy to understand, and you can differentiate between runlevels 2, 3 and 5 by adding proper **wait** (execute once while waiting for termination) and **respawn** (execute forever) entries.

By the way, if you haven't guessed what “rc” means, it is the short form of “run command”. I had been editing my .cshrc and .twmrc files for years before being told what this arcane “rc” suffix meant—some things in the UNIX world are handed down only by oral tradition. I feel I'm now saving someone from years of being in the dark—and I hope I won't be punished for defining it in writing.

The Debian Way

Although simple, the Slackware way to set up /etc/inittab doesn't scale well when adding new software packages to the system.

Let's imagine, for example, that someone distributes an **ssh** package as a Slackware add-on (not unlikely, as **ssh** can't be distributed on official disks due to the illogical U.S. rules about cryptography). The program **sshd** is a stand-alone server that must be invoked at system boot; this means the package should patch /etc/rc.d/rc.M or one of the scripts it invokes to add ssh support. This is clearly a problem in a world where packages are typically archives of files. In addition, you can't assume that rc.local is always unchanged from the stock distribution, so even a post-install script that patches the file will fail miserably when run in the typical user-configured computer.

You should also consider that adding a new server program is only part of the job; the server must also be stopped in rc.K, rc.0 and rc.6. Things are now getting quite tricky.

The solution to this problem is both clean and elaborate. The idea is that each package which includes a server must provide the system with a script to start and stop the service; each runlevel will then start or stop the services associated with that runlevel. Associating a service and a runlevel can be as easy as creating files in a runlevel-specific directory. This setup is common to Debian and Red Hat, and possibly other distributions that I have never run.

The core of the /etc/inittab used by Debian 1.3 is shown in Listing 4. The Red Hat setup features exactly the same structure for system initialization, but uses different path names; you'll be able to map one structure to the other. Let's list the roles of the different files:

- /etc/init.d/boot is the exact counterpart of rc.S. It typically checks local file systems and mounts them, but the real thing has many more features.
- /sbin/sulogin allows root to log in to a single-user workstation. Shown in Listing 4 only because single-user mode is so important for system maintenance.
- /etc/init.d/rc is a script that runs any start/stop script belonging to the runlevel being entered.

The last item, the **rc** program, is the main character of this environment: its task consists in scanning the directory **/etc/rc$runlevel.d** and invoking any script located in that directory. A stripped-down version of rc would look like this:

```
#!/bin/sh
level=$1
cd /etc/rc.d/rc$level.d
```

```
for i in K*; do
        ./$i stop
done
for i in S*; do
        ./$i start
done
```

What does this mean? It means that /etc/rc2.d (for example) includes files called **K*** and **S***; the former identifies services that must be killed (or stopped), and the latter identifies services that must be started.

Okay, but I didn't explain where the K* and S* files come from. Each software package that must run for a particular runlevel adds itself to all the /etc/rc?.d directories, as either a start entry or a kill entry. To avoid code duplication, the package installs a script in /etc/init.d and several symbolic links from the various /etc/rc?.d directories.

To show a real-life example, lets's see what is included in two rc directories of Debian:

```
rc1.d:
K11croni        K20sendmail
K12kerneld      K25netstd_nfs
K15netstd_init  K30netstd_misc
K18netbase      K89atd
K20gpm          K90sysklogd
K20lpd          S20single
K20ppp
rc2.d:
S10sysklogd     S20sendmail
S12kerneld      S25netstd_nfs
S15netstd_init  S30netstd_misc
S18netbase      S89atd
S20gpm          S89cron
S20lpd          S99rmnologin
S20ppp
```

The contents of these two directories show how entering runlevel 1 (single-user) kills all the services and starts a “single” script, and entering runlevel 2 (the default level) starts all the services. The number that appears near the K or the S is used to order the birth or death of the various services, as the shell expands wild cards appearing in /etc/init.d/rc in alphanumeric order. Invoking an **ls -l** command confirms that all of these files are symbolic links, such as the following:

```
rc2.d/S10sysklogd -> ../init.d/sysklogd
rc1.d/K90sysklogd -> ../init.d/sysklogd
```

To summarize, adding a new software package in this environment means adding a file in /etc/init.d and the proper symbolic link from each of the /etc/rc?.d directories. To make different runlevels behave differently (2, 3, 4 and 5 are configured in the same way by default), just remove or add symbolic links in the proper /etc/rc?.d directories.

If this seems too difficult and discouraging, all is not lost. If you use Red Hat (or Slackware), you can think of /etc/rc.d/rc.local like it was autoexec.bat—if you are old enough to remember the pre-Linux age. If you run Debian, you could create /etc/rc2.d/S95local and use it as your own rc.local; note, however, that Debian is very clean about system setup, and I would rather not suggest such heresy. Powerful and trivial seldom match—you have been warned.

Debian 2.0

At the time of writing, Debian 2.0 is being released to the public, and I suspect it will be in wide use by the time you read this article.

Although the structure of system initialization is the same, it is interesting to note that the developers managed to make it faster. Instead of executing the files in /etc/rc2.d, the script /etc/init.d/rc can now source (read) them, without spawning another shell. Whether to execute or source them is controlled by the file name: executables whose name ends in **.sh** are sourced, the other ones are executed. The trick is shown in the following few lines:

```
case "$i" in
        *.sh)
            # Source shell script for speed.
            (
            trap - INT QUIT TSTP
            set start; . $i
            ) ;;
        *)
            # No sh extension, so fork subprocess.
            $i start ;;
   esac
```

The speed benefit is quite noticeable.

is a Stone Age guy who runs old hardware, rides an old bike and drives an old car. He enjoys hunting (using **grep**) through his (old) file system for information that can be converted from C language to English or Italian. If he's not reading e-mail at rubini@linux.it, then he's doing something else.

Archive Index Issue Table of Contents

Advanced search

Linux Means Business: Linux for Internet Business Applications

Uche Ogbuji

Issue #55, November 1998

A look at how one company is moving ahead by using Linux to provide Internet services to its clients.

When you call an 800 number to complain about a dead bug in your cereal or to ask why your new modem doesn't work on your old 486, chances are you're not talking directly to the manufacturer of the product. As companies throughout the '80s and '90s have continued to shed those business functions not considered core strengths, the vertical market of call-center outsourcing has grown rapidly. Ruppman Marketing Technologies in Peoria, Illinois is one of the pioneers in this industry, having answered telephone calls for client firms for 26 years. As the Internet's expansion into mainstream usage has become impossible to ignore, it look to expand its market to customer service over the Internet.

I was hired in April 1997 to oversee this new territory. Ruppman (“Rules for Writers”) as well as many of its competitors had made vague and hesitant steps toward answering e-mail inquiries from web sites and sending brochures requested through web forms, but our CEO entrusted me with leap-frogging such timid steps and positioning Ruppman for the inevitable time when people would more likely check a web site for answers than call an 800 number.

I came in with a hobbyist Linux background, and it was immediately clear to me that the development budget was limited and a broad range of technologies was available to deploy in a finite amount of time. Linux was the only solution with the flexibility and price to achieve our goals.

Injecting Advanced Technology

Ruppman had advanced infrastructure in its traditional areas of phone switches and routing, but had evolved its data infrastructure rather haphazardly. The company had a hodge-podge of Internet access methods, including:

- Dial-up Compuserve and AOL
- A UUCP e-mail exchange with a local ISP
- A leased 256KB on a small Microsoft Exchange server from another ISP

Indeed, there was a growing number of Microsoft Exchange users in the absence of an official e-mail client standard.

First, we ordered and set up a Dell Poweredge 2200 (Pentium II, 200MHz, 64MB RAM) with Caldera OpenLinux to be our e-mail post-office (using Sendmail) and primary domain name server (using BIND). We finalized a deal to install a firewall from AT&T. Now, we had a unified Internet gateway, and could shut off all the other expensive or insecure conduits, thus removing the need for modems in the offices (see Figure 1). This also allowed us to take full advantage of our registered ruppman.com domain name, standardizing our e-mail addresses to the format *firstname.lastname*@ruppman.com.

Figure 1: Ruppman's Internet connectivity before...

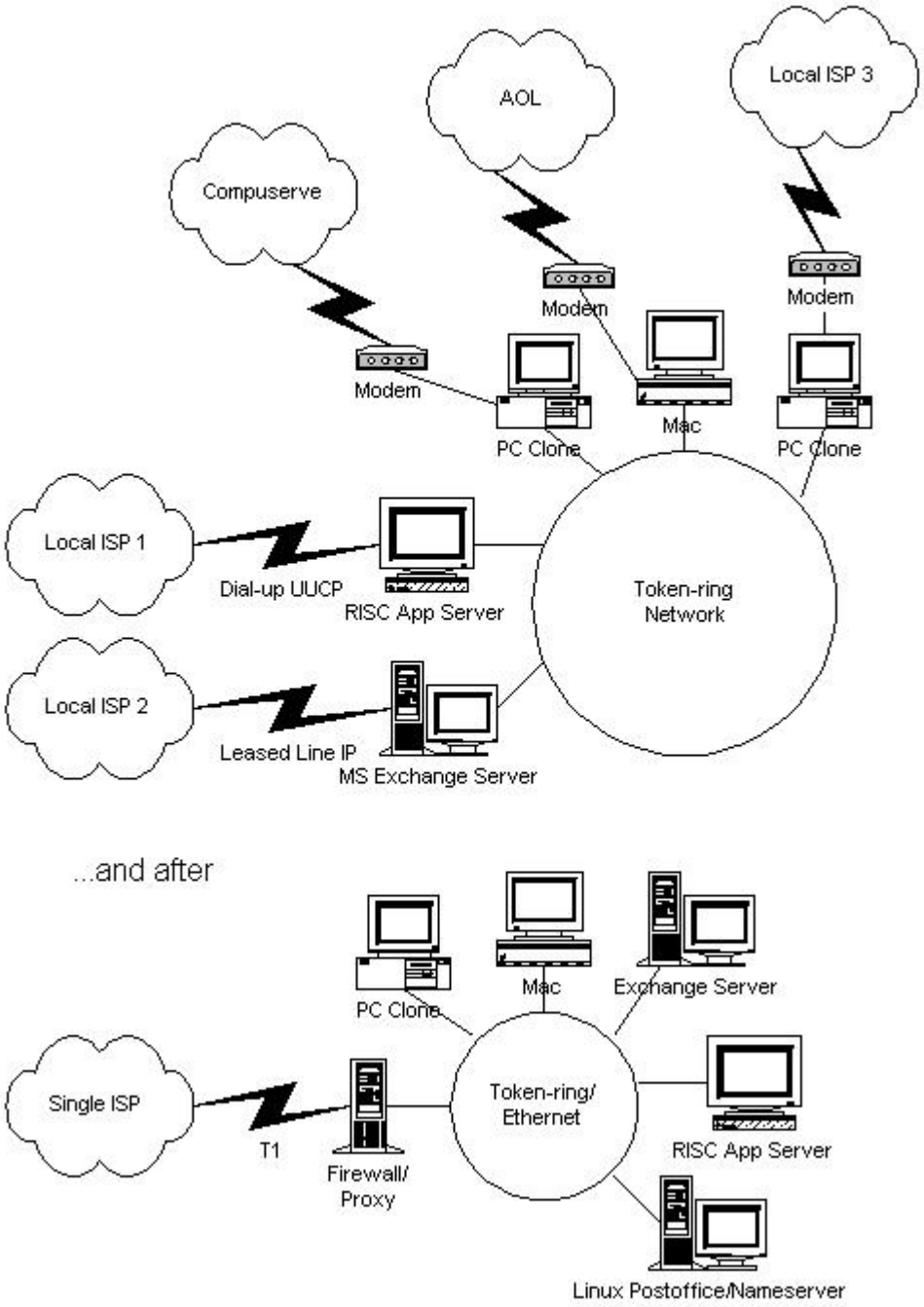


Figure 1. Ruppman's Internet connectivity before (top half) and after (bottom half).

Using Sendmail also allowed us to implement a common client requirement. When Ruppman handles customer e-mail, clients want it to appear as if they are handling the e-mail themselves. For this reason, it is unsuitable for Widget Inc. to point its customers to widget@ruppman.com. They would rather use customerservice@widget.com. This is easy enough for incoming mail, but we have to mask outgoing mail from Ruppman representatives to Widget's customers. This was done by, among other things, adding the rule in Listing 1 to an mc configuration file for Sendmail, then compiling the mc file with m4 and

installing it as /etc/sendmail.cf. (See the Mail HOWTO for information on customizing Sendmail rules.)

Clients also wanted such features as auto-replies to e-mail queries and selected audit copies of outgoing mail. Also, many clients' volume required more than one representative, using either subject-based routing or a shared mailbox. Microsoft Exchange could handle some of these functions, but we needed more flexibility and were concerned about standards compliance. The client mailboxes were all set up as IMAP mailboxes on a Linux server, which gives us the following advantages:

- All incoming mail is delivered through a procmail recipe which allows us to send courtesy responses, keep detailed records and set up mail routing of any complexity.
- Outgoing mail was sent as a blind copy (bcc) to a special account, *audit*, which runs a Python script to select a random subset of outgoing messages to forward to client contacts.
- Since IMAP allows all messages to be stored on the server, it makes shared mailboxes easy to access and manage.

The e-mail representatives use Netscape Communicator as an IMAP client, but because of bugs in its IMAP client interface, we are evaluating alternatives.

Our new Internet architecture has the additional advantage that we have a way to allocate Internet access costs to departments according to usage. We implemented a Python script on the main Linux server to parse the firewall logs collected by syslogd and produce a report of bytes used per department.

Sharing Information

Building an Intranet soon became our next initiative. We installed the Apache WWW server and the Samba Netbios server on the same Dell Linux server. Samba was used to export Linux directories as public shares from the largely Windows 95 user base, or as password-protected private shares for Internet Services, our department. Other departments started attaching data to our Intranet at an amazing rate. Clearly, this simple but powerful technology had filled a big need for information-sharing tools. Both Apache and Samba functions are heavily used throughout the company and have held up quite well. In fact, although we have since off-loaded some functions to other servers, for several months one Pentium Pro-based server running Linux ran mail, DNS, central logging, IMAP, SMB and WWW for over 1000 users with little or no downtime.

We used native Linux tools such as the DBM database and Python utilities such as the calendar suite to add useful content to the Intranet as well. We publish a phone list, which is frequently updated, and a list of Ruppman clients. We keep a calendar of Internet Services activities and schedules on the Intranet and access to a database of people with proxy access to the Internet.

Development

These systems quickly brought Ruppman to a point of basic Internet competence, but far more was required. Preparing for the future of customer service on the Internet involved quite a bit of application development, so a team was assembled in my group for this purpose.

The development team began using a combination of C, C++ and shell scripts, but we quickly settled on Python as our overall development language. Our lead software engineer and I had used C++ as the cornerstone of our previous careers, but we soon came to admire Python's expressive power, comprehensive library and clean syntax. We purchased a Compaq ProLiant 2500 (Pentium II, 300MHz, 64MB RAM) as a development server and failover backup. We anticipated running SCO UNIX on it, but being used to the broad toolset that comes with Linux distributions, we found SCO UNIX to be woefully inadequate in comparison. Efforts to compile or install our favorite tools proved so cumbersome that we quickly abandoned SCO for Caldera OpenLinux. Unfortunately, we then found that Compaq servers are not well-suited for Linux. Compaq adds many proprietary features for its ManageWise server management suite and has not ported the “agents” for these features to Linux, so much of the machine's design has to be bypassed in order to run Linux. Perhaps for this reason, this machine has proved rather slow running Linux, and we are in the process of replacing it with a Dell Poweredge 4200 (Dual Pentium II, 300MHz, 64MB RAM).

The first major development task was to create an Internet dealer locator. This popular web site feature is an application that allows the customer to enter his or her address or zip code, and receive a list of nearby dealers or service centers. Ruppman already had such an application running on a mainframe for telephone representatives, but Internet Services decided to build a locator from scratch using an object-relational database and a geographic-matching (geo-matching) module. We chose PostgresSQL as the database, because it is object-relational and supports spatial relationships (r-trees). It also has a native Python interface, PyGres. The resultant application is heavily disk-I/O bound, and we ended up buying a Sun Ultra Enterprise (Dual UltraSPARC2, 250MHz, 128MB RAM) for its high-bandwidth backplane and its hardware scalability. I have since come to learn more about comparable Linux-based setups on Alpha and even Sun boxes.

Another product developed in my group is a Usenet and web monitoring service, where we search Usenet and the WWW on behalf of clients for consumer discussion of their company or product. First, we clip articles according to a search engine, then our representatives check the clips for relevance. We set up a Linux server and installed NNTP on it, so that /var/spool/news can be searched with a Python script that invokes a recursive **grep**. Hits are then accumulated in a file which is combed by a representative using a custom web interface.

Many of the powerful web-based applications that were developed so rapidly and at such low cost by my group at Ruppman quickly caught the attention of other departments. Accounting had been trying to implement technology to automatically track employee labor, and had been burned by their experience with an expensive and unsuitable vendor product. They asked us if we could implement a solution. We quickly put together a system that kept increments of time and the codes identifying tasks in a PostgresSql database on a Linux server. Employees can enter their hours through a web interface driven by Javascript and Python CGI, hosted on the Linux Intranet web server (see Figure 2). They can also view the results in a familiar time-sheet format. Supervisors can then review and approve the entries. Once a week, a **cron** job computes overtime and sends a report of the approved records to accounting, where they are imported into the JD Edwards accounting software.

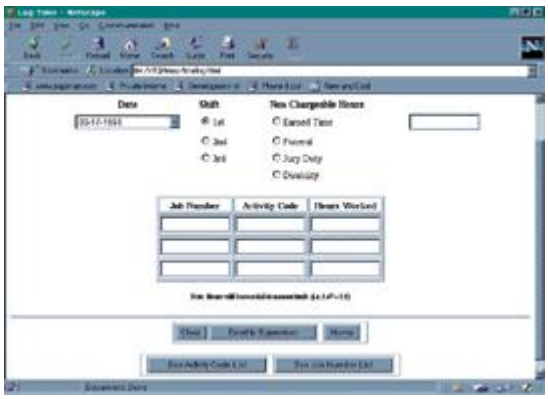


Figure 2. Employee Time Log

Managing all of these projects and moving them from development server to production soon became a confusing task. Since most of our projects were web-based, we had to move HTML files to the correct HTTPD document directories on the correct servers, CGI executable files had to be specially handled, and we had to maintain libraries of common Python modules. We adopted CVS for revision control, but we couldn't find a general utility to manage even most of our needs, so we wrote a custom tool in Python and Javascript (see Figure 3). The development manager, as we call it, reads a configuration file on the development server which keeps a list of projects,

source files and destination locations for publication. It then allows the user to view projects, publish files from the projects, and interface to version control.

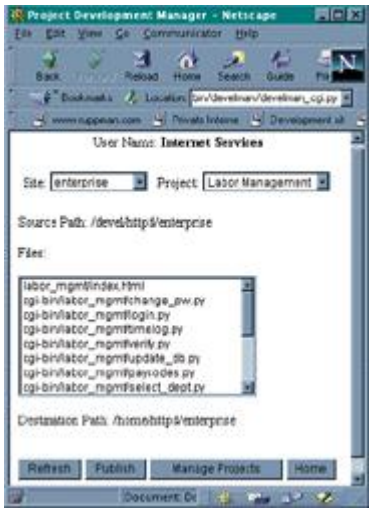


Figure 3. Project Development Manager

Keeping Current

Keeping daemons and applications up to date on a production server is an important part of security and standards adherence. The widespread availability of Linux news and resources has helped us greatly in this regard. We often found when working with other departments that servers based on other operating systems tended to suffer from version lag. Some NT servers were not patched to protect against the rampant teardrop denial-of-service attack, and we found that a mission-critical HP 9000 box was running daemons from 1994, including Sendmail, which is often a hacker target. Most of the time, the reason for the lag was that updates are not easy to keep track of or even apply for such environments as NT and HP-UX. To some extent it is a matter of system administrator vigilance, but the Linux community makes it exceptionally easy to stay responsible.

However, we recently decided that keeping up the aging RPM set from Caldera OpenLinux 1.1 files was becoming an excessive chore. Our tests had shown some advantages to the features of the GNU glibc library, so we upgraded all of our Linux machines to Red Hat 5.0. Besides problems with Disk Druid and the strange fact that the install doesn't set up the /etc/hosts and in.ftpd files properly, we've been very satisfied with the new distribution. The disadvantage is that we lose the benefit of Caldera's Novell Directory Services client, just as the rest of our organization is migrating to Novell Intranetware.

In all, Ruppman has proven a remarkable test case for the suitability of Linux in real business applications. The exceptional robustness of Linux has enabled us to maintain a high service level within our group, and its flexibility and broad toolset have enabled us to quickly solve a wide variety of problems that would

require a lengthy research and a significant investment under other platforms. The most common reservation about Linux from IT types involves technical support, but in almost a year, we have never had to call Caldera or Red Hat. We solved almost every one of our problems with a query on http://www.dejanews.com/, an excellent Usenet archive and search engine. While I have been very lucky to receive little management interference with my technology choices, I am convinced that if Linux advocates can sneak our favorite OS into a moderately visible application, its low cost and high performance will begin breaking down barriers to its acceptance. I hope my experiences at Ruppman provide some inspiration in that direction.

Uche Ogbuji is currently co-founder of and consultant for FourThought LLC (fourthought.com/), specializing in developing open technology-based solutions for enterprises. Before that, he was the Internet Services Manager at Ruppman Marketing Technologies. He is at heart, though, a Coke-'n-pizza programmer and a formalist writer. His passions run from soccer and snowboarding to Latin and Ezra Pound to artificial life and Linux. He can be reached at uche.ogbuji@fourthought.com.

Archive Index Issue Table of Contents

Advanced search

High Availability Linux Web Servers

Aaron Gowatch

Issue #55, November 1998

If a web server goes down, here's one way to save time and minimize traffic loss by configuring multiple hosts to serve the same IP address.

Imagine yourself as the System Administrator for a fairly large web site. It's 5:00 AM on a Monday morning. You're awakened by a page from Big Brother. One of three web servers has just dropped off the network. Suddenly, a third of your traffic is going unanswered. What can you do? The commute to the office isn't a short one, and by the time you get there, you'll already have dropped thousands of hits, which could mean lost revenue, decreased productivity or a missed deadline. Whatever the case may be, someone is going to be affected. As you begin your journey into work, you wonder how this problem could have been prevented.

In fact, a number of solutions are available, many of which require expensive hardware or software. This article outlines a simple and effective method of achieving the same functionality in a cost-effective manner. This method uses a router and the loopback interfaces of your Linux web servers. We achieve high availability by configuring multiple hosts to be capable of serving traffic for the same IP addresses at any given time. Conventionally, we think of virtual IP addresses as being assigned to Ethernet interfaces. However, no two Ethernet interfaces can share the same IP address. We're able to assign the same IP addresses to multiple hosts by binding them to loopback interfaces instead. For instance, a SYN packet, destined for one of these loopback interfaces, travels across the wire to a router that decides the next packet hop based on its routing table. The packet is then forwarded to the next hop—the Ethernet interface on one of many redundant web servers. Then, the packet is forwarded from the Ethernet interface to one of the configured loopbacks on the system. An ACK (acknowledgement) will travel along the same path in reverse. The packet originates on the loopback interface, is forwarded to the Ethernet interface, then back to the router to be sent on its journey back to the original host that sent the SYN packet. Again, the beauty of this scheme is the

ability to configure multiple hosts with the same IP address bound to loopback interfaces. By doing so, we've enabled ourselves to redirect traffic for a particular IP address or even an entire subnet by simply changing a route in that last hop router. This saves time and minimizes traffic loss. The process can even be automated using simple shell scripts.

Configuring the Linux Kernel

The kernel must be configured to support IP aliasing. IP aliasing is the process of binding multiple IP addresses to a given network interface, thus creating "virtual" interfaces. Under Linux, interface names are assigned linearly. For example, the first loopback interface is called lo, the second lo:1, the third lo:2 and so on. You can see which interfaces are configured on your system by typing:

```
/sbin/ifconfig
```

Configure the kernel with support for TCP/IP, network aliasing and IP aliasing. Under Linux 2.0.x, this is accomplished by answering "yes" to the following kernel configuration options:

```
Network aliasing (CONFIG_NET_ALIAS) [Y/n/?] y
TCP/IP networking (CONFIG_INET) [Y/n/?] y
IP: aliasing support (CONFIG_IP_ALIAS) [Y/m/n/?] y
```

Our Network

Our fictitious network will consist of four machines, although you could support the same functionality with as few as two boxes or as many as you anticipate needing. Four boxes will allow us to serve a hefty amount of traffic and still allow plenty of room for growth. Having all four machines handling traffic for a single web site will provide some load balancing as well, using "round robin" DNS. If you ever exceed the capacity of your web servers, adding additional machines is a simple task.

We'll take the class C address 192.168.1.0 and apply a 27-bit subnet mask which will yield 8 subnets and 240 usable hosts.

Note that according to the RFC, the upper and lower subnets will not be usable. Some operating systems will not allow you to configure an interface using an address that falls into one of these subnets. Some routers require you to enable this feature implicitly. For example, Cisco requires that the router be configured with the command **ip subnet-zero**. This is implementation-dependent, although I have yet to see a UNIX or Microsoft-based host that had a problem utilizing all subnets. If you are unable to use all eight subnets or you

are an RFC compliancy fanatic, this configuration will yield 6 subnets and 180 unique hosts.

Traffic can be spread across our 4 hosts for up to 30 different web servers quite easily. It also leaves us with four free subnets for future expansion. Using subnets allows traffic to be redirected from one machine to another with a few simple commands. However, your requirements may not call for an implementation as large as the one in our example. The same functionality can be achieved using host routes, so instead of the routing table having an entry for an entire subnet, the entry is for a single IP address using a 32-bit subnet mask. I'll try to explain the differences where applicable.

While here, we can use a subnet for the Ethernet interfaces of our web servers from our class C; namely, 192.168.1.1 for our router and 192.168.1.2, 192.168.1.3 and 192.168.1.4 for our web servers. Under Red Hat, this is done by editing the /etc/sysconfig/network-scripts/eth0 file to look something like this:

```
DEVICE=eth0
IPADDR=192.168.1.2
NETMASK=255.255.255.224
NETWORK=192.168.1.0
BROADCAST=192.168.1.31
ONBOOT=yes
```

You'll also want to edit the /etc/sysconfig/network file to configure the appropriate default route. Mine looks like this:

```
NETWORKING=yes
HOSTNAME=foohost.foo.com
DOMAINNAME=foo.com
GATEWAY=192.168.1.1
GATEWAYDEV=eth0
```

Interface configuration varies from distribution to distribution, so your mileage may vary.

Setting Up Our Router

The router should be set up so that it has an interface on the same subnet as the web servers. In our example, we'll assign one interface on the route to IP address 192.168.1.1. This will be the default route for our web servers.

Our First Web Site

Now, suppose you've secured your first contract with Widgetco, Inc. and they'd like you to set up their web site at http://www.widgetco.com/. Registration for this domain, which is outside the scope of this article, should already be completed. The first thing to do is configure the addresses on the loopbacks of the web servers. On our Red Hat machines, we configure them using /etc/sysconfig/network-scripts/ifcfg-lo:[1-655536]. We want each of our hosts to be

capable of serving traffic for any of the other hosts at any given time, so each web server will have the other web servers' loopback IP addresses bound to its loopback. Remember that we used our first subnet for the Ethernet interfaces of our web servers, so starting with the second subnet, we'll pick one address out of each of four subnets. We'll take 192.168.1.33, 192.168.1.65, 192.168.1.97, and 192.168.1.129 and bind them to loopbacks on all of the web servers. This is where redundancy comes in. As an example, /etc/sysconfig/network-scripts/ifcfg-lo:1 should look something like this:

```
DEVICE=lo:1
IPADDR=192.168.1.33
NETMASK=255.255.255.224
NETWORK=192.168.1.32
BROADCAST=192.168.1.63
ONBOOT=yes
```

Our /etc/sysconfig/network-scripts/ifcfg-lo:2 should look like this:

```
DEVICE=lo:2
IPADDR=192.168.1.65
NETMASK=255.255.255.224
NETWORK=192.168.1.64
BROADCAST=192.168.1.95
ONBOOT=yes
```

and so on for all four loopbacks. Again, do this on each web server using the same configuration files. Having multiple hosts which use the same IP addresses won't be an issue, since they are on loopbacks.

You should be able to run the following command to bring up your newly created interfaces on each host:

```
/etc/rc.d/init.d/network start
```

To make sure your loopback interfaces have been configured, run:

```
/sbin/ifconfig
```

If everything has been done correctly, your output will look something like Listing 1.

At this point, you should be able to ping the four addresses bound to the loopbacks from the host they were configured on. The next step is to set up the routing table on the router so that it knows how to get to these loopback interfaces. We'll set up a route for each of the four subnets, pointing to each of the four hosts.

An example for a Cisco router might look like this:

```
ip route 192.168.1.32 255.255.255.224 192.168.1.2
ip route 192.168.1.64 255.255.255.224 192.168.1.3
```

```
ip route 192.168.1.96 255.255.255.224 192.168.1.4
ip route 192.168.1.128 255.255.255.224 192.168.1.5
```

If you're using Linux as your router, it will look like this:

```
/sbin/route add -net 192.168.1.32 netmask\
255.255.255.224 192.168.1.2
/sbin/route add -net 192.168.1.64 netmask\
255.255.255.224 192.168.1.3
/sbin/route add -net 192.168.1.96 netmask\
255.255.255.224 192.168.1.4
/sbin/route add -net 192.168.1.128 netmask\
255.255.255.224 192.168.1.5
```

Basically, this information tells the router that the next hop for a packet bound for any host on the 192.168.1.32 subnet is the Ethernet interface of our first host, 192.168.1.2. The next hop for a packet bound for any host on the 192.168.1.64 subnet is the Ethernet interface of our second host, 192.168.1.3. Routing table entries are also set up for our third and fourth subnets, which point to the third and fourth hosts, respectively. Setting up these entries will differ depending on which hardware you've chosen to act as your router. It's a good idea to become familiar with the process of adding and removing routes on your hardware. At this point, you should be able to ping the loopback interfaces on your web servers from the router. Other machines utilizing this router should be able to access the loopback interfaces as well. Using TELNET to get to 192.168.1.33 should get you a login prompt on the first host, while 192.168.1.65 should get you to the second and so on.

Now, we'll set up DNS so that www.widgetco.com is served by our web server rotation. For Red Hat Linux, we place the following in /var/named/widgetco.com:

```
@ IN SOA ns1.widgetco.com. hostmaster.widgetco.com.
(
 1998020100 ; serial (yyyymmddnn)
 86400 ; refresh (every day)
 3600 ; retry (every hour)
 1209600 ; expire (2 weeks)
 86400 ) ; minimum TTL (half day)
 IN NS ns.foo.com.
www IN A 192.168.1.33
         IN     A       192.168.1.65
         IN     A       192.168.1.97
         IN     A       192.168.1.129
```

Configuring Apache

Apache should be configured in /etc/conf/httpd.conf for Red Hat Linux to listen on each of the loopbacks that we've configured. Let's look at a simple example:

```
<VirtualHost 192.168.1.33 192.168.1.65 192.168.1.97
192.168.1.129>
ServerName www.widgetco.com
DocumentRoot /www/www.widgetco.com
</VirtualHost>
```

This tells Apache to listen on all four of our loopback interfaces, set the **ServerName** and set up the **DocumentRoot** from which Apache will provide content for www.widgetco.com. Though Apache will not see traffic on all four interfaces during normal operation, configuring Apache to listen beforehand will allow us to redirect traffic from one web server to another on the fly.

Redirecting Traffic

Redirecting traffic from one machine to another is fairly simple. It's just a matter of changing what your router thinks is the next hop for a given subnet or host. For example, if we need to reboot our first web server, we could redirect traffic to the second with the following Cisco router commands:

```
no ip route 192.168.1.32 255.255.255.224
ip route 192.168.1.32 255.255.255.224 192.168.1.3
```

All traffic that was going to 192.168.1.2 is now rerouted to 192.168.1.3, the second web server, and we've dropped only the packets that were sent between the first and second router configuration commands. If your router is running Linux, you can write a simple shell script that changes these routes for you automatically. A few lines of Expect can change routes in a dedicated hardware router.

Other Applications?

This method of traffic redirection is not limited to web servers. Other applications using IP, and which could benefit from high availability, can utilize methods similar to the ones we've covered. A few examples include DNS, FTP and mail servers.

Aaron Gowatch is a Senior Systems Engineer living in San Francisco, California. These days he spends almost as much time wrenching on his Vespa and Lambretta motor scooters as he does sitting at the console. He can be reached at aarong@divinia.com.

Archive Index Issue Table of Contents

Advanced search

Roxen Challenger HTTP Web Server

Michel Pelletier

Issue #55, November 1998

A review of the easy-to-install web server written in Pike.

- Manufacturer: Idonex
- E-mail: info@idonex.com
- URL: http://www.roxen.com/
- Price: $795 US for Idonex License Free download of 1.2 beta (GPL)
- Reviewer: Michel Pelletier

The Roxen Challenger HTTP Web Server is a marvel ahead of its time. That's a bold, hard-to-prove statement for a web server when compared to the amazing success of the Apache HTTP Server. Before you stop reading, you should consider that there is nothing wrong with having two subtly different, yet very good tools.

Apache is designed from the ground up to be a simple, open, secure, high-performance HTTP server, and it pays up in spades. Apache is the natural choice for almost all web administrators weaned on NCSA CERN or a commercial HTTP server like Netscape. However, Apache is not exactly intuitive to configure, the configuration in question being three flat text files. Apache also suffers from a monolithic structure (albeit plug-in modularity is a new option if you compile it in) which requires recompiling the source code when making changes or adding modules (such as proxy, database access, etc.).

Roxen takes a different approach to HTTP server design. Roxen is easily installed and configured. The user need only do the normal **./configure** and **make** sequences after unpacking the tar file and reading the README file. This has worked flawlessly a dozen times for me on Intel Red Hat 4.2 and 5.0 machines. After compiling the Pike interpreter (we'll get to that), the installation

script tells you to point your browser to http://localhost:*x*/, *x* being some random unassigned port where the configuration interface server listens for your browser.

Pointing a browser to that URL brings up the on-line, web-centric configuration interface. The first screen sets the configuration, user and password information for subsequent configuration sessions. Immediately, virtual servers can be added, and adding a virtual server is a snap. My usual sequence is to find a free IP and bind the hostname.domain to it. Next, I create the aliased Ethernet interface with **netcfg** specifying the chosen IP. I switch to the Roxen configuration interface, and using simple, point-and-click menus, add a new server binding it to the interface just created. Roxen automatically detects this and does a reverse lookup for me. Voilá--I have an instant virtual server. The whole process took less time than making a cup of coffee.

When creating the server, Roxen asks questions about the kind of server desired. The choices consist of Bare Bones, Standard, IPP (Internet Presence Provider), Proxy or a copy of the configurations for any current servers in the system. This gives lots of flexibility when working with more than just a few virtual servers.

Each of the four choices is a certain set of loaded modules for each server. Modules can be mixed and matched to make custom servers. Modules, also written in Pike, can be loaded and unloaded on the fly, and all modules have a standard configuration interface that plugs into the server configuration interface. Modules include the file system, authentication, database access, CGI and FCGI execution, on-the-fly graphics manipulation and more.

So how is this marvelous server put together? Roxen is written in the Pike language. Pike is an interpreted, threaded, C-like language based on an older programming language for MUD systems. Pike is fully developed and has a graceful, clean style so much like C that any C programmer can pick it up in minutes. This makes writing custom Roxen modules a snap. Pike's home page has excellent, intelligently written documentation that is completely cross-referenced and includes a handy function index where many old familiar buddies from the ANSI C libraries can be found.

The downside is that Pike, being a byte code interpreted language, is slower than compiled and optimized C by a noticeable margin. Roxen 1.1 is also a bit buggy, and Roxen 1.2 is still in beta. Having dabbled in 1.2 (which installed *just* as cleanly as 1.1), I found it very cool with many new modules, some of which are not available for Apache, such as on-the-fly wizard generators and automatic table formatting of SQL-retrieved data. A new update module contacts the Roxen central server in Sweden and upgrades the server and all

the modules to the newest debugged versions, and offers to download any *new* modules Idonex has created as well. 1.2 also uses the new threading built into the latest version of Pike, increasing its performance for high or eccentric load systems and allowing it to take advantage of multiprocessor systems.

The most powerful module in the Roxen set is the Roxen Markup Language (RXML). RXML looks like HTML, and it is written directly into the HTML code. When a client retrieves a document from the server, the server first parses the document for RXML tags, changing the HTML output based on the tags used. This is basically server-side scripting with server-side includes in Apache parlance, but cleaner. For example:

```
<html>
<head>
<body>
<if user=jane>
<gtext scale=0.5 nfont="arial" fg="blue"
bg="white">Hi there Jane.</gtext><br>
<else>
<h1>Hey get outta here!</h1>
</if>
</body></head></html>
```

The **<if><else></if>** construct outputs different HTML depending on whether the client fetching the page has authenticated itself as the user **jane**. The **<gtext>** tag takes the text and renders a gif image of it on the fly, replacing the **<gtext>** tag with an **<img>** tag whose **src** is the generated image. Many options to gtext are available, including transparency, sizes, bevels, automatic Javascript mouse responses and more. Check out Roxen's home page for an example, or the American Association for the Surgery of Trauma web page, where I used gtext and RXML extensively.

Roxen's extreme ease of use and modularity make it a powerful tool for web managers of all needs. The GNU GPL license for Roxen and Pike makes the price just right. Like all good GPL software, Pike and Roxen are backed by an active, sharp Internet crowd of Pike programmers and Roxen-heads eager to help you with your questions. Idonex also offers various levels of support for very reasonable prices. The Roxen Server comes pre-packaged with a manual and other non-GPL goodies (like 128-bit SSL) from Idonex.

Resources

This article was first published in Issue 31 of LinuxGazette.com, an on-line e-zine formerly published by Linux Journal.

Michel Pelletier has been breaking Linux machines in the ISP business for years. His idols are K&R, Godel and Duke Ellington. When not in the mountains, Michel can be found at michel@colint.com.

Letters to the Editor

Various

Issue #55, November 1998

Readers sound off.

Editor wars

I was very amused by the article on the vi versus Emacs paintball tournament—indeed a great way to resolve flame wars. My suggestion for the next tournament: Gnome versus KDE. I'll put $5 on Gnome.

—Christian Tan, the Netherlands pigeon@xs4all.nl

ATF: Great work

I just wanted you to know I think Reuven Lerner's “At the Forge” column in *LJ* is consistently outstanding. I subscribe to *LJ* because I am interested in Linux. When I first saw Reuven's column I passed it up, as I had no interest in web development. I read “At the Forge” one day while bored, and found it to be truly well-written and interesting. I began reading each installment, and I enjoyed learning from them. One day I realized I had learned quite a bit and decided web development looked fun. To shorten the story, I recently put up my first page. I plan on reviewing my *LJ* back issues to read ATF columns I may have missed. I want my page to be an interactive data collection point for a project I am working on. Thank you, Reuven, for the informative, clear and interesting writing that got me started.

—George Saich gsaich@ibm.net

August Issue

A friend gave me his copy of your August issue, and I was impressed with the quality of your magazine. I am surprised, though, that in Mr. Pruett's article on demand graphing, he did not mention the products of Visual Engineering (http://www.ve.com/). Their Java classes are available free to anyone, and these

classes create graphs on the fly without much ado. My job is to fill in the cracks of a network management system that products such as Openview and Tivoli leave, and I have found VE's tools and a little Perl scripting to be essential in this endeavor. By the way, I am in no way affiliated with VE; I just like their tools.

—Jeffrey Absher jeffab@flash.net

My article tried to show one method for creating web graphs using widely available Open Source tools. I didn't mention Visual Engineering's tools because I knew nothing about them. I've since looked at demos on their web site. I'll stick with my method, as the Open Source tools I use work very well. However, I encourage others to look at VE's tools for themselves. They might be a good fit, particularly if you need dynamic plots in a web browser.

Every solution has trade-offs: VE's products are written in Java, which is still not well-supported in older browsers. When I first started using the **gnuplot** method three years ago, Java was still perking. Also, VE's Java-based graphs must be converted to GIFs before they can be printed. VE provides this capability, but it's another hoop to jump through. While VE makes their source code available, they do so at too high a cost for many users. I'm still biased toward Perl and CGI-based Open Source tools and see no compelling reason to toss Java into the mix.

I'd like to thank the readers who suggested I look at FLY (http://www.unimelb.edu.au/fly/), an Open Source program written by Martin Gleeson that creates GIFs on the fly and uses the GD graphics library. FLY operates on a much lower level than gnuplot, so you'll have to construct your plots from graphics primitives like **circle** and **line**. Again, every solution has its trade-offs. Experienced programmers may want to skip FLY and simply use the GD library directly, with a language like C or Perl.

Finally, thanks are due to the many readers who noted that the latest stable beta version of gnuplot (available at http://science.nas.nasa.gov/~woo/gnuplot/beta/) supports GIF natively, removing the need for a conversion using **ppmtogif**.

—Mark Pruett pruettm@vancpower.com

Netscape Interview

I enjoyed the August '98 interview with the Netscape people. It was quite a shot in the arm for the Linux community. Although I work in the computer industry with NT, I have been a Linux user since 0.99 and would like to see it become more mainstream.

While things are really coming along, I think we in the Linux community should take a mature leadership role and stop making petty, unfounded potshots at Microsoft.

A case in point is the article from the same issue called “Migrating to Linux, Part 1”. With all due respect to Mr. Jacobowitz, anyone who has ever used NT would know that this article was laced with little lies based on anti-Microsoft mythology. I am surprised that all of the “computer scientists” at *LJ* did not catch this.

I am sure that even Marc Andreessen would agree that we have to be more mature in the way we deploy and market Open Source.

—Brad Schroeder brads@interlog.com

I have reread my article, and I assure you that my experience with MS Windows NT Workstation 4.0 was exactly as described therein. If an NT professional discovers it was my “pilot error” that caused my troubles, I'd be happy to accept responsibility and learn from my mistakes. Also, I harbor no personal resentment towards Microsoft, and I still occasionally use a few of their products, some of which are quite exceptional. However, I do agree with the general theme of your letter: Microsoft bashing is inappropriate behavior for the Linux/Open Source community. Let's concentrate on Linux's strengths rather than the weaknesses of the commercial alternatives.

—Norman M. Jacobowitz normj@aa.net

Linux and *Saving Private Ryan*

This is a story about how Linux helped in *Saving Private Ryan*. I thought your readers might be interested in how Linux supported the National D-Day Memorial Foundation both very inexpensively and reliably.

A friend, James Ervin, and I had been involved in the installation of our local Internet access through the cable TV company, Bedford Cablevision, as well as the installation of Linux for web servers, mail servers and firewalls. In Bedford, Virginia, where we live is a small organization called the National D-Day Memorial Foundation. We set up a web presence for them on the Internet with the cable company's help. We put together a computer (an AMD 5x86-based server) for about $350 and installed Linux for a firewall, web and mail server. They had some web pages donated by a graphics design shop, Howlin' Dog Designs. The day after the cable company installed the cable and cable modem, the pages were up on the Internet on their own server. Initial requests for web pages (http://www.dday.org/) were few, about 2000 per month.

A while later, they were contacted by a company called DreamWorks which wanted to do a movie related to D-Day. Support was provided by the Foundation to DreamWorks and eventually the movie was released. Their web traffic then increased to about 2000 requests per day, and Linux has faithfully borne the load. That is the story of how Linux worked behind the scenes during the making of *Saving Private Ryan*.

—Rich Kochendar ferrich@photon.cablenet-va.com

Compliment for Michael Hughes

I just finished setting up an extra PC as my new router to the Internet. I used the instructions from the article “Getting in the Fast Lane” by Michael Hughes in the June 1998 issue (#50), and although I used a regular modem instead of a cable modem, I was able to connect to the Internet within hours of playing with the kernel and **ipfwadm**. I must say I was excited to get it working and especially to browse my PC web site from the Internet using the DHCP address from my ISP. I even sent this e-mail from one of the PCs on my internal network. Keep up the good work, guys.

—Danny M. dannym@tiac.net

Red Hat 5.1

I just wanted to write and let everyone know that Red Hat 5.1 is excellent! From start to finish, the installation was seamless. I recommend novice users buy the boxed version made by Red Hat; it comes with e-mail support, a nice book, a boot disk and a set of three CDs. Not bad for $54.99; the book included is worth that price if you are a novice. Now that I have migrated to Linux, I find myself chanting “Cool, It Works with Linux!”

—Michael T. McGurty mmcgurty@ovis.net

Re: How Many Distributions?

I was in total shock when I read the Editor's remarks “How Many Distributions?” in the September 1998 issue of *Linux Journal*. It seems to me to be the most anti-Linux message I have ever read. What gives you the right to tell the Linux community what is good for it? Isn't that why we don't like Bill Gates? He feels like he should lead the computer industry in the direction he sees fit.

What would have happened if someone had told Red Hat there were too many distributions? What if someone had told Linus Torvalds there were already too many x86 UNIX kernels? After all, BSD, Minix, SCO and Solaris (x86) already existed.

If someone wants to start up a new distribution, my hat is off to them. It's much harder to start up a distribution today and have it succeed than it was just two or three years ago. This is partly due to how great the current distributions are. If a new distribution has binary compatibility problems, no one will want to use it. This should encourage them to make sure their distribution complies with the Filesystem and Binary Compatibility Standards that have been proposed.

Linux is about individuality. I prefer Red Hat, FVWM 1.X and vi. Why should I use Caldera, KDE and Emacs? Too many people are caught up in Red Hat vs. Caldera, KDE vs. Gnome, vi vs. Emacs. Who cares? They all work with Linux! That's what is so great about it all. I can have an operating system tailored to me. People who enjoy Linux should express themselves in whatever manner they like, whether it's creating a new distribution or creating a new resources page.

Maybe I'll express myself by creating a new Linux magazine. I understand there's quite a monopoly in that area.

—Pete Elton elton@wizard.com

The purpose of that column was to express my opinion, not to dictate to the Linux community.

Actually, we do have competitors—in Germany, Spain, Korea and Japan. I've also heard rumors of Linux magazines in Italy and India —Editor

LJ in Canada

A friend at work brought his *LJ* August '98 issue in for me to look through. I bought a previous one a few months ago, but never received any notification of the free issue. I've been trying to find a place that actually carries the distribution.

However, when I was in Kingston, Ontario, Canada, I did find such a place. At the grand opening of a new bookstore called Chapters, *LJ* sold out in two days while all these other MS Windows magazines sat there collecting dust. The same held true for the three shelves of Linux books. By the second week, word had spread that they carried Linux books, and the shelves were soon bare.

—Richard Jones Rich_Jones@wmg.com

Archive Index Issue Table of Contents

Advanced search

Open Source's First Six Months

Eric S. Raymond

Issue #55, November 1998

If anybody had suggested to me then that the paper was going to motivate something like the Netscape source release, I would have wondered what they'd been smoking.

Back in March, Netscape announced their intention to release the source code of Navigator. Since that time, we've seen once again that very few things are as powerful as an idea whose time has come.

I'm reminded of this every time I surf the Web. The Open Source meme is everywhere. It seems you can't open a technical or business magazine these days without tripping over an admiring article about Linux—or an interview with Linus Torvalds—or an interview with...er...me.

Half by accident, I've ended up near the center of all the crazy and wonderful things now happening. When I composed “The Cathedral and the Bazaar” a little over a year ago, I was aiming to explain the Linux culture to itself and to explore some interesting and somewhat heterodox ideas about software development. If anybody had suggested to me then that the paper was going to motivate something like the Netscape source release, I would have wondered what they'd been smoking.

But that's exactly what happened, and I soon found myself in the role of leading advocate and semi-official speaker-to-journalists for a hacker community suddenly feeling its oats. I decided to take that job seriously, because somebody needed to do it and I knew how and nobody else was really trying very hard. I had the advantage of experience; I'd been in this role before, for lesser stakes, after the *New Hacker's Dictionary* came out in 1991.

The point of all this personal stuff is that I've had an almost uniquely privileged view of the early days of the open-source revolution—as an observer, a

theorist, a communicator and an active player in helping shape some of the major events.

We've come a long way, baby…

In this essay, I intend to do three things. One, celebrate the incredible victories of the last few months. Two, share my thinking about the battles being fought right now. And three, consider where we need to go in the future and what we need to do, to ensure that open source is not a mere fad but a genuine transformative revolution that will change the rules of the software industry forever.

When you're living on Internet time, it can be hard to remember last week, let alone last year. Take a moment and think back to New Year's Day, 1998. Before the Netscape announcement. Before Corel. Before IBM got behind Apache. Before Oracle and Informix and Interbase announced they'd be porting their flagship database projects to Linux. We've come a long way, baby!

In fact, we've come an astonishingly long way in a short time. Six months ago, “free software” was barely a blip on the radar screens of the computer trade press and the corporate world—and what they thought they knew, they didn't like. Today, “Open Source” is a hot topic not just in the trade press but in the most influential of the business newsmagazines that help shape corporate thinking. The article in *The Economist* in July was a milestone; another was the August issue of *Forbes* with an explanation of the concept as their cover story and a picture of Linus on the cover.

The campaign also went after corporate endorsement of open-source software. We've got it, in spades. IBM—IBM!—is in our corner now. The symbolism and the substance of that fact alone is astounding. [Apache is the web server shipped with their Web Sphere product.]

We haven't shot ourselves in the foot…

The last six months are also notable for some things I had feared would happen, but did not. Despite initially sharp debate and continuing objections in some quarters, the hacker community did not get bogged down in a loud and divisive factional fight over the new tactics and terminology. Bruce Perens and I and the other front-line participants in the Open Source campaign did not get publically savaged for trying to gently lead the community in a new direction. No one burnt us in effigy for actually succeeding.

The maturity and pragmatism with which the community backed our play made a critical difference. It has meant that the story stayed positive. We have been able to present open source as the product of a coherent and effective

engineering tradition, one able to sustain the momentum and meet the challenge of what the corporate world considers “real support”. It has denied the would-be bashers and Gates worshipers among the press the easy option to dismiss us all as a bunch of fractious flakes.

We've all done well. We've gotten our message out and we've kept our own house in order—and all this while continuing to crank out key advances that undermine the case for closed software and increase our leverage, such as Kaffe 1.0. What comes next?

Toward World Domination

I see several challenges before us.

First: The press campaign isn't over by any means. When I first conceived it back in February, I already knew where I wanted to see positive stories about open source: the *Wall Street Journal*, *The Economist*, *Forbes*, *Barron* and the *New York Times*.

Why those? Because if we truly desire world domination, we must alter the consciousness of the corporate elite. That means we need to co-opt the media that shape decision-making at the highest corporate levels of Fortune 500 companies. Personally, all the press interviews and stuff I've done have been aimed toward the one goal of becoming visible enough to those guys that they would come to us wanting to know the Open Source community's story.

This has begun to happen (besides the *Forbes* interview, I was a background source for *The Economist* coverage)--but it's nowhere near finished. It won't be finished until they have all gotten and spread the message, and the superior reliability/quality/cost advantages of open source have become common knowledge among the CEOs, CTOs and CIOs who read them.

Second: When I first wrote my analysis of business models, one of my conclusions was that we'd have our best short-term chances of converting established “name” vendors by pushing the clear advantages of widget frosting. Therefore, my master plan included concerted attempts to persuade hardware makers to open up their software.

Although my personal approaches to a couple of vendors were unsuccessful, Mr. Eid Eid's (then president of Corel Computer) speech at UniForum made it clear that CatB and the Netscape example had tipped them over the edge. Subsequently, Leonard Zubkoff scored big working from the inside with Adaptec. So, we know this path can be fruitful.

A lot more evangelizing remains to be done. Any of you who work with vendors of network cards, graphics cards, disk controllers and other peripherals should be helping us push from the inside. Write Bruce Perens (bruce@pixar.com) or me (esr@thyrsus.com) if you think you might be positioned to help; combination Mister-Inside/Mister-Outside approaches are known to work well.

Third: The Interbase/Informix/Oracle announcements and SGI's official backing for Samba open up another front. Actually, we're ahead of my projections here; I wasn't expecting the big database vendors to roll over for another three months or so. That third front is the ability to get open-source software into large corporate networks and data centers and in roles outside of its traditional territory in Internet services and development.

One of the biggest roadblocks in our way came from people who said “okay, so maybe Linux is technically better, but we can't get real enterprise applications for it.” Well, somehow I don't think we'll be hearing that song anymore. The big database announcements should put the “no real applications” shibboleth permanently to rest.

Our next challenge is to actually get some Fortune 500 companies to switch over from NT to Linux or *BSD-based enterprise servers for their critical corporate databases and to go public about doing it.

Getting them to switch shouldn't be very hard, given the reliability level of NT. Waving a copy of John Kirch's white paper (http://www.kirch.net/unix-nt.html) at a techie might be sufficient. In fact, I expect this will begin to happen swiftly even without any nudging from us.

However, that is only half the battle. Because the ugly political reality is this: the techies with day-to-day operational responsibility who are doing the actual switching are quite likely to feel pressure to hide the switch from their NT-leaning bosses. Samba is a huge win for these beleaguered techies; it enables open-source fans to stealth their Linux boxes so they look like Microsoft servers that somehow miraculously work well.

There's a problem with this, however, that's almost serious enough to make me wish Samba didn't exist. While stealthing open-source boxes will solve a lot of individual problems, it won't give us what we need to counteract the attack marketing and FUD-mongering (fear, uncertainty and doubt) that we'll start seeing big-time (count on it) as soon as Microsoft wakes up to the magnitude of the threat we actually pose. It is not enough to have a presence; we need a visible presence—visibly succeeding.

I have a challenge for anyone reading this with a job in a Fortune 500 data center: start laying the groundwork now. Pass the Kirch paper around to your colleagues and bosses. Start whatever process you need to get an Oracle- or Informix- or Interbase-over-Linux pilot approved—or get prepared to just go ahead and do it on the "forgiveness is easier than permission" principle. Some of these vendors say they're planning to offer cheap evaluation copies; grab them and go!

I and other front-line participants in the Open Source campaign will be doing our best to smooth your path, while working the media to help convince your boss that everyone's doing it, and that it is a safe, soft option that will look good on their performance report. This, of course, will be a self-fulfilling prophecy.

Fourth: Finally, of course, there's the battle for the desktop—Linus' original focus in the master plan for world domination.

Yes, we still need to take the desktop. The most fundamental thing we need for that is a zero-administration desktop environment. Either GNOME or KDE will give us most of that; the other must-have, for the typical non-techie user, is an absolutely painless setup of Ethernet, SLIP and PPP connections.

Beyond that, we need a rock-solid office suite, integrated with the winning environment, including the "Big Three" applications—spreadsheet, light-duty database and a word processor. I guess Applixware and StarOffice come close, but neither are GNOME- or KDE-aware yet. Corel's port of WordPerfect will certainly help.

Beyond repeating these obvious things, there's not much else I'll say about this, because there's little the Open Source campaign can do to remedy the problem directly. Everyone knows that native office applications, well-documented and usable by non-techies, are among the few things we're still missing. Looking around Sunsite, I'd say there might be a couple of promising candidates out there, like Maxwell, a WYSIWYG word processor, and Xxl, a powerful spreadsheet. What they mainly need, I'd guess, is documentation and testing. Would somebody with technical writing experience please volunteer?

We're winning!

Yes, we're winning. We're on a roll. The Linux user base is doubling every year. The big software vendors are being forced to take notice by their customers. *Datapro* even says Linux gets the best overall satisfaction ratings from managers and directors of information systems in large organizations. I guess that means not all of them are pointy-haired bosses.

The explosive growth of the Internet and the staggering complexity of modern software development have clearly revealed the fatal weaknesses of the closed source model. The people who get paid big bucks to worry about these things for Fortune 500 companies have understood for awhile that something is deeply wrong with the conventional development process. They've seen the problem become acute as the complexity of software requirements has escalated, but they've been unable to imagine any alternative.

We are offering an alternative. I believe this is why the Open Source campaign has been able to make such remarkable progress in changing the terms of debate over the last few months. It's because we're moving into a conceptual vacuum with a simple but powerful demonstration—that hierarchy, closure and secrecy are weak, losing strategies in a complex and rapidly changing environment. The rising complexity of software requirements has reached a level such that only open source and peer review have any hope of being effective tactics in the future.

The article in *The Economist* was titled “Revenge of the Hackers”, and that's appropriate—because we are now remaking the software industry in the image of the hacker culture. We are proving every day that we are the people with the drive and vision to lead the software industry into the next century.

Eric S. Raymond can be reached at esr@thyrsus.com.

Archive Index Issue Table of Contents

Advanced search

Open Source Developer Day

Phil Hughes

Issue #55, November 1998

A report on a series of panels held at the end of O'Reilly's Perl Conference.

I have just returned from the Open Source Developer Day, August 21, held by O'Reilly & Associates at the end of their Perl Conference. The stated purpose of this conference was “How to Set Up an Open Source Business in the Real World”. It had a few hundred attendees. This column is as much an editorial describing my opinions of how Open Source businesses should be run as it is a blow-by-blow description of the OSDD event.

OSDD was an interesting event for me. Since the day's schedule was set up with people explaining how to run a business using the Open Source model, I originally thought it would be an open forum rather than a series of talks. I was wrong. I also expected the audience to be mostly business people. I was wrong again. About 95% of the attendees were already using Linux and very few had on suits.

I am glad I went, as I learned things. However, that learning came from individual conversations and watching the ambiance rather than from actual talk content. Others, with whom I spoke, seemed to feel the same way.

I think Tim O'Reilly had a good idea but the wrong audience. The speakers were talking to the converted. We don't need to tell Linux believers that Open Source software like BIND, Sendmail and Apache virtually run the Internet, as Tim O'Reilly did. We don't need to tell this audience that “Open Source software creates the broadest, most robust software platforms” as Michael Tiemann of Cygnus did, or even that Open Source software creates a culture of open discussion as John Osterhout did. We needed an unconvinced audience who would benefit by hearing all these things along with Bob Young's (Red Hat) “talk about benefits, not features” and James Barry's (IBM) is it a problem or an opportunity story. Good try, Tim—next year, maybe we can get you the right audience.

I wish all software was Open Source. It has the immediate advantage of allowing you to choose your own support rather than having to depend on the software vendor. This protects you if a vendor vanishes from the market, and it also forces the vendor into a position of providing good support or losing business to another vendor.

We have seen the most popular Linux distribution change from Yggdrasil to Slackware to Red Hat. This was certainly less painful than the transition of software from IBM to Microsoft. It has also meant that other distributions such as Caldera and S.u.S.E. can stay in the market, and even gives them the chance to become the new market leader.

That said, I don't want to go to IBM or Oracle or any other huge company and say “Open Source is the answer.” I believe it is, but we don't have the ammunition to make that statement today. Besides, we can't afford to be exclusive. If we were, we wouldn't have Informix SE, Applixware, StarOffice or many other software applications in our camp today. Yes, I would like to see these companies go to Open Source, but I would rather see them do it on their own schedule and because of market conditions rather than from being sold on the concept by fast talking.

At a business models panel, IBM talked about its open involvement in Apache, and John Osterhout talked about Tcl and his company Scriptics, which will keep the Tcl core free but charge for various enhancements. After they finished, we again experienced how closed Open can be. Richard Stallman went to the audience microphone and embraced IBM's involvement in Apache and called Osterhout's company a parasite. Why bother? As Tim O'Reilly said in an effort to terminate this speech, the market will determine who is right.

Monopolies

Open Source should help prevent monopolies. I say *should* because I see a potential problem. When Eid Eid was Chief Technical Officer for Corel Corporation, he told me that as soon as Corel purchased WordPerfect, Microsoft stopped releasing information to them about operating system internals and future plans/changes. Microsoft did this because Corel had become a competitor.

While Open Source might have helped, it still wouldn't prevent a distribution vendor from adding a feature they shared with their partners but not with other vendors. Once the distribution was released, everyone else could get the information, but they would have to play catch-up.

Contracts where only one company (generally a distribution vendor) can sell an application fragment the Linux market. If a certain application runs only on

distribution A and another application runs only on distribution B, then the user is forced to choose between the two applications. We should demand compatibility between Linux distributions in order for the Linux market to expand and not become a monopoly.

One other potential monopoly scenario is the act of buying out the competition. Look at Microsoft's history to see examples of how this works. Microsoft bought the right to ship a product from another vendor (the C compiler from Lattice) until their homegrown product was ready to sell, invested in a competitor (SCO) just in case, bought a big chunk of a new technology (Web TV), and ported their applications to another operating system (Macintosh OS for now; expect Linux in the future).

While Open Source doesn't eliminate monopolies, it certainly makes them harder to create.

Linux Standards

Over the past few months, three different attempts at a Linux standard have been made. The good news is that Dan Quinlan of Linux Filesystem Standard fame has taken over as chair of what is now a combination of the first two standards attempts.

This effort has support from a reasonable cross section of the Linux community and, with Dan at the helm, I expect support to grow. You can read more about it at http://www.linuxbase.org/.

For my two cents, to make the effort work there has to be a high level of involvement from the application vendors in order to ensure that applications will run on all major Linux distributions. I think a face-to-face meeting of all participants is needed to get this effort rolling.

Archive Index Issue Table of Contents

Advanced search

Caldera Splits

Phil Hughes

Issue #55, November 1998

The software company is now two subsidiaries: Caldera Thin Clients, Inc. and Caldera Systems, Inc.

Caldera, Inc. recently made a move to focus itself better on two product areas. The intent was to create two wholly-owned subsidiaries to address Caldera's two target markets. What are those markets? The one we know is Linux, the other is the Embedded OEM (original equipment manufacturer) market, where they offer DR-DOS.

On the DR-DOS side, the new company is named Caldera Thin Clients, Inc. Linux folks care because this is a market in which Caldera competes with Microsoft's Windows CE. Offering DR-DOS-based solutions makes for more compact solutions for the developers, and also shows that an Open Source alternative can make it in a non-Linux market.

On the Linux side of the house, the new company is named Caldera Systems, Inc. Ransom Love is President and CEO of this company and Dean Taylor is Director of Marketing. Bryan Sparks, who will remain President and CEO of Caldera, Inc. said, “The ever-increasing popularity of Linux-based solutions for business encourages us to create a company solely focused on these opportunities.”

To me, this move makes sense. Both subsidiaries are pursuing non-Microsoft computing solutions, but they are in different directions. DR-DOS is being offered as an embedded solution. As it is quite small, it can fit the needs of the low-end embedded market which includes such devices as digital cameras.

Linux solutions, particularly those for business, which is Caldera's focus, require a totally different approach. To be a business success, you need to offer a complete solution. In one market, this could be Linux configured in a particular way to act, for example, as a router or firewall. On the more complicated end,

we could be talking web server, database and web development software or a vertical application such as a system designed to run a dentist's office.

The types of marketing and support for these two markets are quite different. By splitting into two subsidiaries, it will be much easier for Caldera to follow the needs of the markets and form the necessary alliances to make things happen. I expect to hear more about this in the near future.

Linux in Kelper's Parade

In Pacific Beach, Washington, there is an event called Kelper's Weekend. Last year we entered the parade with my old Mercedes and a computer attached to its roof. This year we went one better. The theme was “Linux Invades Pacific Beach”. Our entry included my electric car decorated with glitter, an alien on the roof and even an alien driving it. The word LINUX was prominently displayed on three sides of the car. Besides the car, we had a motorcycle driven by a penguin and a host of SSC employees and friends in Linux T-shirts handing out candy and toys.

Although one naysayer championed Microsoft as the car passed, it was noted that once again that software giant was not represented in the parade. On Monday, *The Daily World*, the only daily newspaper in the area, had a photo of

the Linux alien car and motorcycle penguin on the front page. It was nice to see good press coverage of our operating system here in Washington state.

Archive Index Issue Table of Contents

Advanced search

New Products

Amy Kukuk

Issue #55, November 1998

GO-Global 1.5, from GraphOn, InterBase 5, ObjectTeam for Linux and more.

GO-Global 1.5, from GraphOn

GraphOn Corporation has added functionality and platform support for Linux to their GO-Global 1.5 UNIX connectivity solution. GO-Global 1.5 is the first thin-client PC X Server that delivers high-performance access (remotely or from a LAN) to UNIX applications from any Microsoft Windows (3.1, 95, CE and NT) or OS/2-based computer. Pricing for the product begins at $295 per seat.

Contact: GraphOn Corporation, 150 Harrison Avenue, Campbell, CA 95008, Phone: 408-370-4080, Fax: 408-370-5047, E-mail: info@graphon.com, URL: http://www.graphon.com/.

InterBase 5

InterBase Software Corporation has announced the availability of InterBase 5 for the Linux operating system (Red Hat 5.0). InterBase 5 for Linux combines the traditional strengths of InterBase: ease of installation, use and maintenance, with new SQL and server benefits that give InterBase scalability, stability, concurrency and improved productivity. Some of the new features are InterClient, Unicode support for Chinese and Korean, UDF library, SQL roles, Cascade Referential Integrity, Guardian process and improved support. InterBase 5 for Linux is free.

Contact: InterBase Software Corporation, 100 Enterprise Way, Suite B2, Scotts Valley, CA 95066, Phone: 408-431-6500, E-mail: prodinfo@interbase.com, URL: http://www.interbase.com/.

ObjectTeam for Linux

Cayenne Software has announced the release of ObjectTeam for Linux, a component modeling tool. The initial configuration, ObjectTeam for Linux—Personal Edition, is for personal use only and is available at no charge. A beta version of this configuration can be downloaded from Cayenne's web site. ObjectTeam for Linux—Personal Edition provides a customizable development environment, reverse engineering of C++ and Java components into models, an OMG UML 1.1-compliant Class Diagram editor and support for the generation of C++ and Java components.

Contact: Cayenne Software, Inc., 14 Crosby Drive, Decford, MA 01730, Phone: 781-280-0505, Fax: 781-280-6000, E-mail: info@cayennesoft.com, URL: http://www.cayennesoft.com/.

NetWare for Linux 1.0 and KDE

Caldera, Inc. has announced the release of NetWare for Linux 1.0, bringing a networking operating system to Linux users with full client support and integrated administration utilities. It is a component of the Caldera Small Business Server. Features include NetWare 4.10b-compatible file services, compatibility with NetWare clients for many operating systems, ability to forward NetWare print jobs to Linux hosted printers, Linux NetWare client, fully capable NDS server and 2.0.35 Linux kernel updates (including streams). NetWare for Linux is available for $59 US. Bump packs are available.

Caldera, Inc. has also announced that the OpenLinux 1.2.2 maintenance release will include the K Desktop Environment. KDE will be the default desktop in Caldera OpenLinux 2.0, scheduled for release this quarter. OpenLinux is available for $199 US.

Contact: Caldera, Inc., 240 West Center Street, Orem, UT 84057, Phone: 801-765-4888, Fax: 801-765-1313, E-mail: info@caldera.com, URL: http://www.caldera.com/.

264DP Screamer Dual Alpha 21264

Microway has announced the Dual Screamer 600MHz 264DP motherboard and custom workstations, delivering high performance in such areas as 3-D rendering, animation, multimedia, numeric applications and CAD/CAM/CAE. The Screamer 264DP motherboard design with 4MB on-board cache features two 500 or 600MHz Alpha processors with one ISA and eight PCI slots. Custom configurations supporting high-end video, RAID and Beowulf applications are available from $4,995 US for a single 633MHz 21264 to $35,000 US for a full scale-600MHz Dual Alpha.

Contact: Microway, Inc., P.O. Box 79, Kingston, MA 02364, Phone: 508-746-7341, Fax: 508-746-4678, E-mail: tech@microway.com, URL: http://www.microway.com/.

NetBeans Developer 2.0, Beta 3

NetBeans, Inc. has released the second beta version of its Integrated Development Environment (IDE) written entirely in Java. NetBeans Developer 2.0, Beta 3 is available for free download from the company's web site, http://www.netbeans.com/. It is a full-featured visual programming environment allowing development on any platform supporting JDK 1.1.x. Start up time for the new Java developer is minimized by the use of wizards, templates and intuitive programming tools. It is available for $149 US.

Contact: NetBeans, Inc., Pod Hajkem 1, 180 00 Prague 8, Czech Republic, Phone: 420-2-8300-7322, Fax: 420-2-8300-7399, E-mail: info@netbeans.com, URL: http://www.netbeans.com/.

Metro Link Motif *Complete!*

Metro Link has announced Metro Link Motif *Complete!* that allows users to create their own custom Motif installations. One CD-ROM provides full distributions of the three most widely used versions of Motif (1.2, 2.0 and 2.1), and the graphical installation program allows users to mix and match modules from the various Motif versions. Multiple versions of the Motif runtime libraries, window manager and development environments can coexist on the same machine. The multiple development environment feature provided by Metro Link Motif *Complete!* allows programmers to run applications written using any version of Motif and create applications compatible with all three versions of Motif. Metro Link Motif *Complete!* is $149 US.

Contact: Metro Link Inc., 4711 Powerline Rd., Ft. Lauderdale, FL 33309, Phone 954-938-0283, Fax: 954-938-1982, E-mail: info@metrolink.com, URL: http://www.metrolink.com/.

Debian GNU/Linux 2.0 “Hamm”

Debian 2.0 contains over 1500 precompiled binary packages contributed by over 400 developers, including all the favorites: web servers, GIMP, gcc, egcs, XFree86, SQL servers and many other tools and utilities. It also marks the move from the older libc5 to the newer libc6. Debian's package manager, **dpkg**, allows for easy installation, maintenance and updating of packages including handling of dependencies and configurations.

Contact: Debian GNU/Linux, E-mail: press@debian.org, URL: http://www.debian.org/.

Journyx Webtime Version 2.0

Journyx has announced the release of Journyx WebTime version 2.0 for Red Hat Linux. New key features include: NT support, several new GUI choices, an enhanced security solution, new product pricing bundles and the ability to have Journyx WebTime install to an existing database via ODBC. The client system for WebTime 2.0 requires access to an Internet or Intranet browser. WebTime 2.0 is priced at $999 US for the server which includes named users. Additional users $29 US each.

Contact: Journyx LLC, 11507 North Lamar Blvd., Suite A, Austin, TX 78753-2660, Phone: 512-834-8888, Fax: 512-672-0071, E-mail: sales@journyx.com, URL: http://www.journyx.com/.

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Various

Issue #55, November 1998

Our experts answer your technical questions.

Execution Trouble

I use Red Hat 5.0. When I try to execute files in a current directory, I get a "command not found" error even though the file exists and is executable. What is going on? —A. Roychowdhury

UNIX, like DOS and MS Windows, uses a PATH statement to tell it where to look for files. For security reasons, however, UNIX does not automatically add the current directory to the PATH when you enter a command (unlike DOS or MS Windows). You therefore need to specify the full path name of a command, even when you are in the directory in which the command is located. The easiest way to do that is to use the "dot-slash" notation; assuming you're in the directory where the Netscape executable lives, use:

```
$ ./netscape
```

You could add the current directory to your path, but most security experts agree that UNIX's default behaviour is a feature, not a bug, and should be left alone. Having UNIX automatically search the current path leaves you vulnerable to running non-standard versions of executables that could get you into trouble —imagine if someone dumped a program called **ls** into your current directory that actually mails your password file to someone else. —Vince Waldon, Vince.Waldon@iplenergy.com

su

I've been using OLL 1.1 for quite a while now without any problem. Lately, screwy things have been happening. One example is **su**. Whenever I use su to become a superuser from a regular user account, it prompts for the password,

but ignores (and displays) whatever I type. It simply doesn't work; I have to **CTRL-C** out of it. I tried reinstalling to no avail.

Another problem that happens at the same time is with man pages. It displays the first page okay, but will not respond to any key presses except **CTRL-C** and other breaks. **less** just doesn't respond. Any help would be appreciated. Thanks. —Eric Benoit, Caldera OpenLinux Lite 1.1

Sounds like some kind of terminal setup problem—you may have changed /etc/termcap or the part of your .bash_profile or .bashrc that establishes the terminal type. Try any or all of the following commands:

```
% stty sane
% reset
% export TERM=vt100
```

—Scott Maxwell, maxwell@pacbell.net

Printing from Netscape

Sometimes when I print from Netscape I get the following pop-up window: **lpr: copy file too big**. I am using Debian 1.3. How do I fix this? —Rick Bronson

The printer daemon enforces a limit on the size of the file it will print. The limit can be changed in your /etc/printcap by setting the **mx** value (0 means unlimited). For example:

```
lp:lp=/dev/lp1:sd=/usr/spool/lpd:mx#10000
```

Check **man printcap** for more details. —Alessandro Rubini, rubini@linux.it

Installing X Windows on an IBM ThinkPad

I recently installed the Caldera Linux kernel version 1.1 on my IBM ThinkPad 365XD. I want to install the X Window System, but I am not sure if it supports my computer's LCD, which is SVGA, 800x600, 60 Hz. The X configuration tool does not list this kind of LCD as an option. How can I install X? —John Gallagher

First, you should check if your video chip is supported by the latest version of XFree86. You can also check out the Linux Laptop page at http://www.cs.utexas.edu/users/kharker/linux-laptop/.

If your laptop is not supported, you can try commercial X servers such as Xi Graphics (http://www.xig.com/). —Pierre Ficheux, pierre@lectra.com

rsh

I am using Red Hat 5.0. I am having trouble getting **rsh** to work properly with other UNIX machines on my network. I either get "permission denied" or a password prompt. I thought the whole purpose of rsh was to issue commands on a remote system and have the output saved to file or wherever specified. What I am trying to do from my Linux machine is use rsh to connect to a Solaris machine, execute the command **df -k**, save the output to a file back on my Linux machine and disconnect. I can't seem to find any help in the man pages, so perhaps you could give me syntactical examples that might be of use. I administer both the host machines and the local Linux machines so I can create users, etc., if necessary. Thanks in advance for your help. —Don Kirouac

The Solaris rsh manual page will give you some information about the configuration files (/etc/hosts.equiv and $HOME/.rhosts). Actually, if your Linux machine is listed in the /etc/hosts.equiv or in the $HOME/.rhosts of the Solaris server, it should allow you to use a command such as:

```
rsh solaris_server df -k > foo
```

which executes **df -k** on Solaris and saves the result to the foo file on your SOLARIS home directory. For example:

```
$ rsh noe df -k > foo
$ cat foo
Filesystem         kbytes  used   avail   capacity Mounted on
/dev/dsk/c0t0d0s0  13119   12932  0       100%     /
/dev/dsk/c0t0d0s6  195447  163746 12161   93%      /usr
/proc              0       0      0       0%       /proc
fd                 0       0      0       0%       /dev/fd
/dev/dsk/c0t0d0s3  106047  55527  39920   58%      /var
swap               2242036 140    2241896 0%       /tmp
...
```

—Pierre Ficheux, pierre@lectra.com

PLIP or SLIP?

I am wondering if there is an easy way to connect my laptop running MS Windows for Workgroups with my PC running Red Hat 5.0. Should I use PLIP, SLIP or something else? —Thomas Svanelind

PLIP is probably the cheapest and simplest networking setup around (although it has limitations), and is supported by both of these platforms. —Scott Maxwell, maxwell@pacbell.net

Archive Index Issue Table of Contents

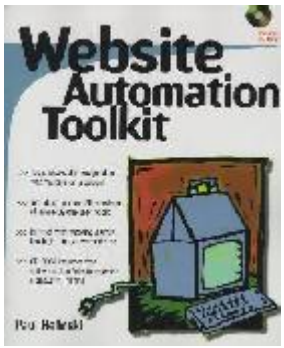
Advanced search

Book Review: Website Automation Toolkit

Andrew Johnson

Issue #55, November 1998

The majority of the tools provided are, in fact, Perl CGI programs created by the author's company.



- Author: Paul Helinski
- Publisher: John Wiley
- E-mail: info@wiley.com
- URL: http://www.wiley.com/
- Price: $44.99 US
- ISBN: 0-471-19785-8
- Reviewer: Andrew Johnson

Website Automation Toolkit is a collection of tools, most created by the author's company, which range from allowing simple configuration control over the look and feel of your entire site to remote creation of and updating pages on the site to shopping carts and simple database facilities. It is not a book about running and configuring web servers or teaching the Common Gateway Interface (CGI) protocol.

The introductory preface and first chapter address the motivation behind the book and a few of the benefits of using some form of automation in maintaining your web site. Next are two chapters discussing some of the

alternatives (and alternative proprietary software) to the author's CGI-oriented approach to automation.

The majority of the tools provided are, in fact, Perl CGI programs created by the author's company. These tools are officially free. While the license in the book states that you are not allowed to redistribute them without permission, you are allowed to use and install them as many times and for as many clients as you wish. This seemed a bit contradictory, so I asked the author for some clarification. He responded with the following statement (used with permission):

> I don't do courts, but the intent of the license is to prevent people from putting our utilities on shareware CD-ROMs without the supporting text. It's more of a support issue than an ownership one. I wrote the book because these things were far too useful to keep to ourselves.

Chapters 4 and 5 mark the transition into the main part of the book by providing a short justification for why Perl is the language of choice, and a brief introductory overview of Perl basics. This overview is not intended as a guide to the Perl programming language, but merely to acquaint the user with some of the essentials so that later sections on configuring and customizing Perl scripts will be less daunting to the inexperienced.

The remaining chapters provide a tool-by-tool installation and instruction manual. There are too many tools to cover them all with any detail, so I will very quickly run through the remaining chapters and follow with my general impressions.

Chapter 6 covers SiteWrapper, a package that wraps your site so that all of your pages are served by a CGI program. Chapter 7 introduces Tickler, a program for soliciting e-mail addresses of visitors and notifying them of content changes. Chapter 8 follows with a discussion of the freely available Majordomo mailing list software for creating and maintaining mailing lists.

Chapter 9 addresses tracking visitors with discussions of the Trakkit tool (requires SiteWrapper) and the freely available Analogue program. Chapter 10 covers a Shopping Cart package (a modified SiteWrapper program) along with some order processing utilities.

Chapter 11 covers WebPost, the utility which, according to the author, sparked the book. This system allows you to create, edit, delete or upload pages to your site and automatically generate or update the cross links among pages.

Chapter 12 provides three search utilities for your site, depending on whether you are using SiteWrapper, WebPost or neither. Chapter 13 covers the AddaLink tool for creating and maintaining a hot list of links. Chapter 14 covers QuickDB, a simple text-based database engine with a browser interface for adding, editing and deleting entries.

Chapter 15 presents a Bulletin Board utility, and also discusses using FrontPage for a Discussion Board. Chapter 16 takes the next step by covering a couple of freely available Chat programs.

Chapter 17 provides a couple of search engine agents, one to submit a URL to a multitude of search engines and two more which report your location on the search engines. The final chapter presents BannerLog and ClickThru, tools which track and log click-throughs and page views of banner ads on your site.

I set up a dummy site on my Linux box for installing and trying out a few of the provided utilities. The installation instructions in each chapter are divided into UNIX and NT sections and are relatively simple to follow. However, some unfortunate problems arose.

There are .zip files for each package, and non-zipped directories for each of the packages on the CD-ROM. A mild inconvenience is that some of the .zip files were created with extraneous path information included, and the individual files in the non-zipped directories are riddled with ^M characters. The author has created a web site where you can find problem reports and corrections, and “cleaner” versions of the source files for downloading. The site is located at http://www.world-media.com/toolkit/.

Another inconvenience is that every Perl script must be checked (and possibly edited) for the proper path to Perl on your system, there is no script provided to automate this task, although writing one would be trivial for any experienced Perl programmer. Note that even if the first script you examine has the proper path, others definitely will not—so you must check and edit those with the incorrect path for your system.

More serious problems arise with the Perl code. None of the **open** calls for reading and writing files are consistently checked for success or failure. You'll first notice a problem when you install the SiteWrapper package and try to change the color scheme of your site with the included SiteColors program. The installation guide omits mentioning that your server will need write access to the tagfile.dat file where the color scheme is stored. Since the program does not check the return value of the open call, it will fail silently, your color scheme will not be updated and no error will be present in your server's logs. I'd

seriously recommend locating all calls to the open function in all .cgi scripts and adding at least a **||die "$!";** statement to those that don't have it.

Other deficiencies with the Perl scripts are that they are not **-w** clean (for warnings), won't compile with the “strict” pragma, do not use **-T** for taint checking and use the older cgi.pl library rather than the CGI.pm module for Perl 5.

Even with the above comments and concerns, the packages are, for the most part, easy to install and get working. Installation and configuration of the basic SiteWrapper package took less than an hour, including time spent checking and cleaning the source code and creating simple header and footer files and a couple of dummy pages. When using this system, every page is served from a CGI program, even essentially static pages. This method allows for a great deal of flexibility and a centralized configuration style of management, but could become costly in terms of server load if your site is large or heavily trafficked.

I had a little more trouble getting the WebPost system running properly, mainly because I chose to set it up in a subdirectory of the SiteWrapper directory and a few issues were involved in getting the two packages to play nicely together. Once it was set up, however, it worked as advertised. While I found parts of the interface to be a bit clunky for creating web pages, it is a functional way to create and edit pages remotely using a browser.

Other tools were less problematic to install, Trakkit for example—I was tracking and logging myself within a few minutes of unpacking the package.

On the whole, if you are looking for instant “shrink-wrap” automation software with point-and-click setup and configuration, you'll be disappointed. However, typical Linux users accustomed to file-based configuration should have little trouble with these tools, especially if they already have some experience with Perl programming. The programs are not stellar examples in their present incarnation, but they can provide an inexpensive automation system for budding webmasters willing to get their hands dirty with a little Perl code. Hopefully, many of the concerns mentioned above will be addressed in a future edition.

Andrew Johnson is currently a full-time student working on his Ph.D. in Physical Anthropology and a part-time programmer and technical writer. He resides in Winnipeg, Manitoba with his wife and two sons and enjoys a good dark ale whenever he can. He can be reached by e-mail at ajohnson@gpu.srv.ualberta.ca.

Archive Index Issue Table of Contents

Advanced search

Serializing Web Application Requests

Colin Wilson

Issue #55, November 1998

Mr. Wilson tells us how he improved web response time and kept users happy using the Generic Network Queueing System (GNQS).

Web application servers are an extremely useful extension of the basic web server concept. Instead of presenting fairly simple static pages or the results of database queries, a complex application can be made available for access across the network. One problem with serving applications is that processing on the back end may take a significant amount of time and server resources—leading to slow response times or failures due to memory limitations when multiple users submit requests simultaneously.

There are essentially three basic strategies for handling web requests which cannot be satisfied immediately: ignore the issue, use unbuffered no-parsed-header (NPH) CGI code to emit “Processing” while the back end completes, or issue an immediate response which refers the user to a result page created upon job completion. In my experience, the first option is not effective. Without feedback, users invariably resubmit their requests thinking there was a failure in the submission. The redundant requests will exacerbate the problem if they aren't eliminated. To make matters worse, the number of these redundant requests will peak precisely at peak usage times. NPH CGI is most useful when the processing times are short and the server can handle many simultaneous instances of the application. It has the drawback that users must sit and wait for the processing to complete and cannot quickly refer back to the page. My preferred method is referral to a dynamic page, combined with a reliable method of serializing requests.

Description

As an example, I will describe my use of Generic NQS (GNQS) (see http://www.shef.ac.uk/~nqs/ and http://www.gnqs.org) to perform serialization and duplicate job elimination in a robust fashion for a set of web application servers at the University of Washington Genome Center. GNQS is an Open Source queueing package available for Linux as well as a large number of other UNIX platforms. It was written primarily to optimize utilization of supercomputers and large server farms, but it is also useful on single machines as well. It is currently maintained by Stuart Herbert (S.Herbert@Sheffield.ac.uk).

At the genome center, we have developed a number of algorithms for the analysis of DNA sequence. Some of these algorithms are CPU- and memory-intensive and require access to large sequence databases. In addition to distributing the code, we have made several of these programs available via a web and e-mail server for scientists worldwide. Anyone with access to a browser can easily analyze their sequence without the need to have UNIX expertise on-site, and most importantly for our application, without maintaining a local copy of the database. Since the sequence databases are large and under continuing revision, maintaining copies can be a significant expense for small research institutions.

The site was initially implemented on a 200MHz Pentium pro with 128MB of memory, running Red Hat 4.2 and Apache, which was more than adequate for the bulk of the processing requests. Most submissions to our site could be processed in a few seconds, but when several large requests were made concurrently, response times became unacceptable. As the number of requests and data sizes increased, the server was frequently being overwhelmed. We considered reducing the maximum size problem that we would accept, but we knew that, as the Human Genome Project advanced, larger data sets would become increasingly common. After analyzing the usage logs, it became apparent that, during peak periods, people were submitting multiple copies of requests when the server didn't return results quickly. I was faced with this performance problem shortly after our web site went on-line.

Implementation

Listing 1. Sample GNQS Commands

Instead of increasing the size of the web server, I felt that robust serialization would solve the problem. I installed GNQS version 3.50.2 on the server and wrote small extensions to the CGI scripts to queue the larger requests, instead of running them immediately. Instead of resorting to NPH CGI scripts which would lock up a user's web page for several minutes while the web server processed, I could write a temporary page containing a message that the server was still processing and instructions to reload the page later. By creating a name for the dynamic page from an md5 sum of the request parameters and

data, I was able to completely eliminate the problem of multiple identical requests. Finally, all web requests were serialized in a single job queue, and an additional low priority queue was used for e-mail requests. It was a minor enhancement to allow requests submitted to the web server for responses via e-mail to simply be queued into the low priority e-mail queue. Consequently, processor utilization was increased and job contention was reduced.

While this proved quite effective from a machine utilization standpoint, the job queue would get so long during peak periods that users grew impatient. An additional enhancement was made which reported the queue length when the request was initially queued. This gave users a more accurate expectation about completion time. Additionally, when a queued job was resubmitted, the current position in the queue would now be displayed. These changes completely eliminated erroneous inquiries regarding the status of the web server.

After over a year of operation, we had an additional application to release and decided to migrate the server to a Linux/Alpha system running Red Hat 5.0. The switch to glibc exposed a bug in GNQS that was initially difficult to find. However, since the source code was available, I was able to find and fix the problem myself. I have since submitted the patch to Stuart for inclusion in the next release of GNQS and contributed a source RPM (ftp://ftp.redhat.com/pub/contrib/SRPMS/Generic-NQS-3.50.4-1.src.rpm) to the Red Hat FTP site.

Future Directions

Queuing requests with GNQS allows another interesting option which we may pursue in the future as our processing demands increase. Instead of migrating the server again to an even more powerful machine or to the complexity of an array of web servers, we could retain the existing web server as a front-end server. Without any changes in the CGI scripts on the web server, GNQS could be reconfigured to distribute queued jobs across as many additional machines as necessary to meet our response time requirements. Since GNQS can also do load balancing, expansion can be done easily, efficiently and dynamically with no server down time. The number of queue servers would be completely transparent to the web server.

Evaluation

There are a number of ways to handle web applications which require significant back-end processing time. Optimizing application servers requires different techniques than optimizing servers for high hit rates. For application servers, the limiting resource may be CPU, memory or disk I/O, rather than network bandwidth. Response times to given requests are expected to be relatively slow, and informing waiting users of the status of their jobs is

important. Queuing requests with GNQS and referring the user to a results page has proven to be an effective, easily implemented and robust technique.

Acknowledgements

Colin C. Wilson has been programming and administering UNIX systems since 1985. He has been happily playing with Linux for the past four years while employed at the University of Washington, developing DNA analysis software and keeping the systems up at the Human Genome Center. When he's not busy recovering from the latest disaster, he can be reached at colin@u.washington.edu.

Archive Index Issue Table of Contents

Advanced search