The Perl Review

Volume 0 Issue 6 November 1, 2002

Web Access http://www.ThePerlReview.com/ **Email** letters@theperlreview.com **Publisher** brian d foy **Editor** Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff **Copy Editors** Beth Linker, Glenn Maciag, Chris Nandor **Contributors** David H. Adler, Paul Barry, Neil Bauman, brian d foy, Andy Lester, Mike Stok, Betsy Waliszewski

Letters

Send your letters, comments, and suggestions to letters@theperlreview.com

TPR Subscriptions

I'd like to subscribe to *The Perl Review*. However, I prefer not to use PayPal. If you tell me where to send a check or cash, I'd be happy to do so.

– *Gregor Dodson*

brian writes: *A lot of people have said something similar, and at the moment we do not have a better way to take money. Once we get going with a printed version, we will have another way to pay with credit cards and a way to take checks. Right now we want to stay lean and mean which makes us avoid merchant and bank services that charge monthly fees. PayPal is free. We will keep track of everyone who wants to use something other than PayPal and let you know when we have an alternative method. We apologize for the inconvenience.*

About this issue

Mike Stok takes a look at Ruby and likes what he finds, while brian d foy reviews the available Ruby books. Paul Barry, the author of *Programming the Network with Perl* builds a network analyzer in under 100 lines of Perl. brian d foy shows a simple use of XML::RSS, which we actually use on our web site. Oh, and Sarah says "Hi", although she did not think we would really print that.

Perl on the web

The Perl Review
http://www.theperlreview.com – the website for this magazine with information for readers and authors

Use.perl
http://use.perl.org – Perl news and commentary

Correction

We made a mistake typesetting Robby Walkers "Filehandle Ties" article in *The Perl Review* (0, 5). In his Multiplex.pm example, line 8 should have read `print $_ $_[0] for @$self;`. We incorrectly used `$_[1]` and apologize for the confusion.

Write for *TPR*

Have something to say about Perl? *The Perl Review* wants first person accounts about using Perl. If you cannot write a complete article you can write a "Short Note". Want to tell everyone about a book you have read? Write a book review! Were you at a Perl function? Give us a trip report!

We would like to get articles or "Short Notes" on

- Bioinformatics
- Perl internals
- Cute Perl hacks
- Debugging Perl
- Creating modules
- *and more ...*

We also like articles aimed at Perl for beginning programmers. Perl people take for granted some things that never make it into books or get passed on to beginners. Do you have something new Perl programmers should know? Perhaps:

- Using templates
- Using configuration files
- Argument processing
- Deciphering documentation

You can get submission guidelines from our website, http://www.theperlreview.com .

Volunteer for *TPR*

We have not turned into a business yet, so we still rely on the generosity and availability of volunteers. We are currently looking for someone to become our RSS wrangler so we can provide feeds of *The Perl Review*.

Community News

Send us your news stories at news@theperlreview.com .

The Perl Journal survives

http://www.tpj.com

The *Dr. Dobb's* crew from CMP, the publisher that put TPJ in *SysAdmin Magazine*, is saving TPJ by publishing a monthly electronic version starting this month. You can subscribe for $12 a year. Columnists include Simon Cozens and brian d foy.

DoD uses Perl

http://www.egovos.org/pdf/dodfoss.pdf

A recent Mitre report prepared for the United States Department of Defense evaluates the value of open source software to them. Conclusion—the DoD should not stop using Perl.

500th job posted

http://jobs.perl.org

Belcan IT of Cincinnati, Ohio posted the 500th job to the Perl Jobs site. You can get job announcements in email, on the web site, through their RSS feed (see the RSS article in this issue, for example), or the nntp.perl.org newsgroup interface.

Quiz of the Week

perl-qotw-subscribe@plover.com

Mark-Jason Dominus now publishes a weekly Perl quiz with answers posted in succeeding weeks. Email the subscription address to start receiving the quiz. Dominus provides regular and expert versions, although no prize money is to be had in either category.

Clean out CPAN!

http://use.perl.org/ brian_d_foy/journal/8314

The size of CPAN is almost to the point that we will need three CDs to fit it all—1.3Gb currently. Randal Schwartz, however, wrote a script to just mirror the most recent versions of everything (excluding the perl distribution itself, about 25Mb gzipped). The mini-CPAN came out to be 225Mb. brian d foy invented the "Schwartz Factor", or ratio of the real size of CPAN to its effective size, currently hovering around 0.174. He estimates that with the current size, if everyone cleaned out old distributions to get the Schwartz Factor up to 0.4, everything will fit on a single CD again.

New books

Publishers: to have us list your book, send us a note at book_reviews@theperlreview.com.

Essential Blogging
Benjamin Trott, et al.; O'Reilly & Associates;
0-596-00388-9; September 2002

Perl CD BookShelf, 3.0
Larry Wall, et al.; O'Reilly & Associates;
0-596-00389-7; September 2002

LDAP Programming
Clayton Donley; Manning;
1-930110-82-0; October 2002

Embedding Perl in HTML with Mason
Dave Rolsky & Ken Williams
O'Reilly & Associates;
0-596-00225-4; October 2002

Computer Science & Perl Programming: Best of The Perl Journal
Edited by Jon Orwant
O'Reilly & Associates;
0-596-00310-2; November 2002

XML CD Bookshelf
O'Reilly & Associates;
0-596-00335-8; November 2002

Programming Web Services with Perl
Randy J. Ray & Pavel Kulchenko
O'Reilly & Associates;
0-596-00335-8; December 2002

Would you like to review a book? Send your review to book_reviews@theperlreview.com

Perl In the Press

Know of another magazine with regular Perl content? Let us know at letters@theperlreview.com

The Perl Journal – http://www.tpj.com

Linux Magazine – http://www.linux-mag.com/

;login: – http://www.usenix.org/publications/login/

Unix Review – http://www.unixreview.com/

Short Notes

If you have something that you want to show off without writing an entire article, like a cool Perl trick, a module you just released or something happening in the Perl community, send your short note, between 200 and 400 words, to *short_notes@theperlreview.com.*

Perl's True Success is in The Telling

Betsy Waliszewski, betsy@oreilly.com
Product Manager for Perl books

My first day on the job at O'Reilly & Associates was unforgettable. Hired primarily as a Perl advocate, I spent that day at the O'Reilly Open Source Convention in Monterey, California, surrounded by the most influential people in the Perl community. Boy, was I overwhelmed! I managed to learn everyone's contribution to the language (not an easy task), but I came away puzzled.

How was I going to translate all this devotion and enthusiasm for Perl to IT managers? How could I convince the programming world at large that Perl was the real thing?

After tossing around ideas with O'Reilly editor Nat Torkington, we decided to promote Perl through a series of stories about folks who actually use it. We sought case studies showing how programmers working for mainstream companies had used Perl in non-trivial applications that were mission-critical. The goal of these "Perl Success Stories" was to demonstrate that Perl is not a toy language, but an important programming component that the big wheels of commerce could rely on to do a variety of tasks.

It wasn't easy to get these stories in the beginning. Programmers who answered our call for clear-cut examples of how much time, money and pain Perl saved them needed approval from their bosses to release this information. People at O'Reilly who wrote the articles sent drafts back and forth, making sure they were accurate and acceptable without having them turn into puff pieces.

We got some amazing stuff. Our first story talked about Perl's use at the Federal Reserve. Others detailed its use by Amazon, Agilent, UniCredito Italiano (Italy's top bank), and the U.S. Census Bureau. Each story chronicled a different—and very critical—use for Perl. Italy's bank used Perl DBI for a data migration from one enterprise data warehouse to another. The Census Bureau employs Perl to display our country's leading financial and economic indicators online, with twice-daily updates from thousands of sources. Perl helped Carnegie-Mellon University develop speech synthesizers for robotics applications. And programmers developed a Perl-based web application quickly to run the Swedish National Pension.

As the stories were done, I posted them on our web site (perl.oreilly.com), and once I'd collected eight of them, I was able to put them together in a booklet for distribution at the Perl 4 Conference. The response was incredible and I received many requests for the booklet afterward. We also sent it to our corporate mailing list.

Three years later, I no longer have to hunt for Perl Success Stories. Programmers now send them directly to me, and you can find the latest on my weblog at www.oreillynet.com (I'm a bit behind in getting them online, but don't worry—they'll all be posted). I also just finished printing the third edition of our collection of Perl Success Stories, a booklet which was distributed at the Perl 6 Conference. We now have more than 25 stories and new ones roll in regularly.

How has corporate America received these stories? Along with other open source tools, Perl is now being taken seriously by IT managers who need a quick development solution—not just for web applications, but as a general purpose language for quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, and networking. The momentum is genuinely building out there.

Do you have a Perl success story? Write it up and send it to me at betsy@oreilly.com, making sure that you cover the following points:

- The company and its business
- The name and function of a the Perl application
- Rough size of the code and its development time
- Rough idea of how heavily the code is used (number of users)

Promoting Perl has been an incredibly fun ride these past three years, and the best part is working with the Perl community. The spirit of sharing and helping each other is unusual and fantastic. I've learned so much about Perl and why it's important. Thank you to all the people who have contributed to this project.

This story is simultaneously published on the O'Reilly website: http://www.oreillynet.com/pub/a/oreilly/perl/2002/11/04/perlsuccess.html

YAPC::Europe trip report

David Adler, dha@panix.com

YAPC::Europe is always an interesting experience for the out-of-towners. We learned that beer, pretzels and sausage really are a huge part of life in Munich.

But that was outside the walls of the Technische Universität München, where we ingested many different flavors of Perl. Unsurprisingly, although this year's conference theme was "The Science of Perl", the subjects ranged widely.

In line with the theme, Karen Pauley talked about psychometric testing and Lucy McWilliams gave a lightning talk on using perl in the study of the sex lives of flies (no, really). Larry Wall gave a variant on his "State of the Onion" talk, using the contents of *Scientific American* as a series of jumping off points for talking about Perl and the community surrounding it. Jos Boumans won the "most talks given" award, with an introduction to object oriented programming, a discussion of his CPANPLUS module and an introduction to POE.

This was the first YAPC::Europe for a number of the big names in Perl. Thanks to a mini-conference in Zurich, Larry Wall, Damian Conway, Allison Randall and Dan Sugalski were able to come to YAPC in Europe for the first time. Needless to say, this was a welcome turn of events.

Much attention was given to the future of Perl, with the Perl 6 team giving various talks on aspects of that project and current Perl 5 Pumpking Hugo van der Sanden not only giving a talk on Perl 5.10, but spending a good deal of time asking people what they wanted to see in it.

Greg McCarroll again demonstrated his ability to get people to empty their wallets for a good cause at the auction by raising 3500 Euros to offset any losses of the conference, with the balance going to The Perl Foundation. Michael G. Schwern received much play in the auction by arm wrestling Damian (he lost) and, finally, auctioning off about half of the clothes he was wearing. It's a long story, but we got money for this!

Days full of Perl, evenings full of social events. Attendees also got a beer krug with the conference logo. Everyone left happy.

News from Geek Cruises

Neil Bauman, neil@geekcruises.com
Somewhere in the Caribbean, October 25, 2002

We are a crew of just about 125 Linux geeks here in the Western Caribbean! I'm sending this, using a wireless high-speed connection, from our ship, Holland America's MS Maasdam.

We're having, of course, a fabulous time. So far, Guido van Rossum's "Introduction to Python" and Ted Tso's "Introduction to the Linux Kernel" have been the most popular. Of course, "Learning Perl", "Introduction to PHP", and "More Than You Ever Wanted to Know About Filesystems" have been quite popular as well.

Earlier today Linus, Guido van Rossum, Eric Raymond, and colleagues conducted a panel discussion and Q & A with the Jamaica Linux Users Group in Ocho Rios, Jamaica. We posed for the camera outside the meeting room after our meeting. Linus is holding two of his daughters; Guido van Rossum is in the center, staring at the camera, with his mouth WIDE open; Eric Raymond is just to the right of me; I'm the bald guy in the catcher's position, in the front row, almost directly below Guido; to the right of me is Steve Oualline, author of *Practical C Programming*, *Practical C++ Programming*, and *Vim – Vi Improved*; to Steve's right is Theodore Tso, a Linux kernel developer since almost the very beginnings of Linux; and our hosts, the members of the Jamaica Linux Users Group.
http://www.geekcruises.com/gifs/ll2/TheRuins.jpg

After our morning meeting a bunch of us went to Dunn's River Falls. What makes this interesting is that you *climb up* the falls, not just hang out enjoying the rushing water. In that photo you see about 20% of the trek.
http://www.geekcruises.com/gifs/ll2/DunnsFall.jpg

Last night Linus held court in the "Card Room"—actually, based on the laughs I'd say it was more like a comedy club than a court room. He was, as usual, blunt and irreverent.
http://205.122.23.229/peng/linusq-a.ogg

[*Editor: GeekCruises has a Perl Whirl in 2003: http://www.geekcruises.com*]

Perl at the MacOS X Conference

brian d foy, comdog@panix.com
Santa Clara, October 1, 2002

O'Reilly & Associates first Mac OS X Conference felt a lot like their first Perl conference—a lot of cool, new things happening in a wide-open field. Apple is still learning a lot of things about the possibilities of Darwin, and even held a secret meeting with lots of big names, including Perl hacker Nat Torkington, to get their opinions on new directions.

On the first morning I gave a tutorial about programming Perl on Mac OS X to highlight the differences, not the similarities. A lot of the audience really knew their Macs and I learned quite a bit from their encyclopedic knowledge of Mac OS X.

That afternoon Dan Sugalski gave an excellent talk about CamelBones which provides Perl hooks into the Cocoa frameworks. In the evening I moderated a Perl discussion that had an amazing group of people show up—Dan Sugalski, Matthias Neeracher (original author of MacPerl and now an Apple employee), Rich Morin of Primetime Freeware, Vicki Brown (co-author of *MacPerl: Power and Ease*), and David Wheeler, maintainer of Bricolage.

Later in the week John Labowitz talked about his Mac::AppleScript::Glue module along with graphics programming with Perl, and David Wheeler talked about moving from Linux to Mac OS X.

Of course, Apple was a major sponsor of the event which works out well for the attendees who want to use the terminal room. It had the latest and most beautiful Apple hardware with their large flat-screen monitors and a friendly Apple technician available.

O'Reilly makes most of the presentations available online. *http://conferences.oreillynet.com.*

Host with pair Networks

- Rock-Solid FreeBSD Unix Reliability
- Edit scripts and compile programs from a secure shell
- Technical, competent support
- Perl 5.6 with over 2000 installed modules
- Free Setup when you switch to us

We've got a web hosting plan waiting for you.

www.pair.com

pair networks

Perl Whirl in Hawaii with
Tim Bunce, Damian Conway, Mark-Jason Dominus, Randal Schwartz, Nat Torkington, Larry Wall, and others
June 1 – June 8, 2003

geekcruises.com
EDUCATION THAT TAKES YOU PLACES

Simple RSS with Perl

brian d foy, comdog@panix.com

Abstract

The Rich Site Summary (RSS) provides a way for web site operators to allow other people to syndicate the web site content. The web site operators publish a *feed* which other people can download, parse, and display on their own site. I present the RSS files that *The Perl Review* provides along with the program we use to parse other site's RSS files. I only show RSS 0.9.

1 Introduction

The Rich Site Summary (RSS) is a set of stories, usually from the same web site. The RSS file contains XML data which other programs can parse to create custom presentations.

There are several major versions of RSS. The first version, 0.9, is simple to read and simple to use. It does not contain as much information as later versions which provide various additions to the RSS format—even arbitrary extension through XML namespace magic. I think that is overkill for my purposes, and I do not want to do the extra work to handle the extra features. I use version 0.9, and I only show that in this article.

The Perl Review publishes RSS files for each issue and for collections of related articles. A few other web sites actually use them. In the other direction, we use several other sites' RSS files on our web site for "Perl at a Glance".

2 Creating RSS

Files in RSS 0.9 typically have the extension `.rdf` for Resource Description Framework. The XML format is very simple. Code listing 1 shows the actual RSS file for the last issue. Line 1 is the required XML header. Line 3 pulls in the appropriate definitions. The RSS data has *channel* and *item* data. The channel is the name of the feed and typically has a title and a link to the original website. Line 7 starts my channel element with a title element with the name of the issue, and a link element to the issue file. The rest of the elements are items, sometimes called *headlines*. Each item has a title and link.

`Code Listing 1: RSS file format`

```
1  <?xml version="1.0"?>
2
3  <rdf:RDF
4  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5  xmlns="http://my.netscape.com/rdf/simple/0.9/">
6
7  <channel>
8  <title>The Perl Review, v0 i5, September 2002</title>
```

```
<link>http://www.theperlreview.com/Issues/The_Perl_Review_0_5.pdf</link>
</channel>

<item>
<title>Extreme Mowing, by Andy Lester</title>
<link>http://www.theperlreview.com/Articles/v0i5/extreme_mowing.pdf</link>
</item>

<item>
<title>What Perl Programmers Should Know About Java, by Beth Linker</title>
<link>http://www.theperlreview.com/Articles/v0i5/perl-java.pdf</link>
</item>

<item>
<title>Filehandle Ties, by Robby Walker</title>
<link>http://www.theperlreview.com/Articles/v0i5/filehandle_ties.pdf</link>
</item>

<item>
<title>The Iterator Design Pattern, by brian d foy</title>
<link>http://www.theperlreview.com/Articles/v0i5/iterators.pdf</link>
</item>

</rdf:RDF>
```

Creating this file is very easy. I can write it by hand, but I can also create it with a program, which I want to do since all of the data comes from a database and I want to automate the process.

Code listing 2 shows a simple program to create the RSS data in code listing 1. The XML::RSS module can handle several versions of RSS, so in line 5 I specify version 0.9. I can change the version number to another one that XML::RSS can handle and see the results. Other versions are a bit more complicated, but XML::RSS handles them for me through the same interface so I can decide to switch later and not have to completely rewrite the script.

On line 10, I create the RSS channel from the first two lines from DATA. On line 15, I start a while loop to process the rest of the data. I skip lines without non-whitespace, chomp the line that I read in the while condition, then read another line of data and chomp that. I expect the first line to be the title and the line after that to be the link address. I add these to my RSS object on line 22.

On line 28, I simply print the RSS data as a string. It should look close to the output in code listing 1, although some people may see slight differences for different versions of XML::RSS.

Code Listing 2: Create RSS files with XML::RSS

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  use XML::RSS;
5  my $rss = XML::RSS->new( version => '0.9' );
6
7  chomp( my $channel_title = <DATA> );
8  chomp( my $channel_link  = <DATA> );
9
```

```
$rss->channel(
                title        => $channel_title,
                link         => $channel_link,
                );

while( defined( my $title = <DATA> ) )
                {
                next unless $title =~ /\S/;
                chomp $title;

                chomp( my $link = <DATA> );

                $rss->add_item(
                                title => $title,
                                link  => $link,
                                );
                }

print $rss->as_string;

__END__
The Perl Review, v0 i5, September 2002
http://www.theperlreview.com/Issues/The_Perl_Review_0_5.pdf

Extreme Mowing, by Andy Lester
http://www.theperlreview.com/Articles/v0i5/extreme_mowing.pdf

What Perl Programmers Should Know About Java, by Beth Linker
http://www.theperlreview.com/Articles/v0i5/perl-java.pdf

Filehandle Ties, by Robby Walker
http://www.theperlreview.com/Articles/v0i5/filehandle_ties.pdf

The Iterator Design Pattern, by brian d foy
http://www.theperlreview.com/Articles/v0i5/iterators.pdf
```

3 Parsing RSS

The XML::RSS module makes parsing RSS even more easy that creating it. In code listing 3, which shows the actual program we use to generate the HTML for “Perl at a Glance”, most of the work deals with HTML, not RSS. Line 7 defines the RSS files to download. I found those by either visiting the site or asking the author if they had RSS files. For instance, Randal Schwartz has RSS feeds for most of his columns although he does not advertise this on his web site—at least not somewhere I could find.

Line 18 defines the location the program stores output files. Each feed has an associated output file which contains just its portion of HTML that another program collates into the final web page.

Line 20 starts the foreach loop which cycles through all of the RSS files. On line 22, I copy the URL to $file so I can manipulate $file and use it as the file name in the open on line 26. If I cannot open the file, I skip to the next feed. On line 34, I select the file handle I just opened so I do not have to specify it in all of the print statements.

On line 36, I create a new RSS object. I do not have to specify the version I want to use because XML::RSS figures it out based on the data I feed it in line 38. Once the module parses the data, I simply access the parts that I need. On lines 40 and 41 I get the channel title and image, which are hash references I store in $channel and $image.

On line 43, I start the HTML output. At some point I will change this program to use Text::Template, but for now something is better than nothing, even if this is horribly wrong. Next issue I will convert this program to store configuration and template data apart from the code to make up for it.

On line 49, I check if $image has a url key. The feed might not have included a logo and I do not want a broken image icon to show up if it did not. With an image, I use the image location to form the link back to the original site, and without an image, I use the channel title.

On line 67, I iterate through the items in the feed and print a link for each one. Once I finish with the items I finish the HTML output and close the filehandle.

Code Listing 3: Fetch and parse RSS feeds

```
 1  #!/usr/bin/perl -w
 2  use strict;
 3
 4  use LWP::Simple;
 5  use XML::RSS;
 6
 7  my @files = qw(
 8  http://use.perl.org/useperl.rss
 9  http://search.cpan.org/rss/search.rss
10  http://jobs.perl.org/rss/standard.rss
11  http://www.perl.com/pace/perlnews.rdf
12  http://www.perlfoundation.org/perl-foundation.rdf
13  http://www.stonehenge.com/merlyn/UnixReview/ur.rss
14  http://www.stonehenge.com/merlyn/WebTechniques/wt.rss
15  http://www.stonehenge.com/merlyn/LinuxMag/lm.rss
16  );
17
18  my $base = ’/usr/home/comdog/TPR/rss-html’;
19
20  foreach my $url ( @files )
21          {
22          my $file = $url;
23
24          $file =~ s|.*/||;
25
26          my $result = open my $fh, "> $base/$file.html";
27
28          unless( $result )
29                  {
30                  warn "Could not open [$file] for writing! $!";
31                  next;
32                  }
33
34          select $fh;
35
36          my $rss = XML::RSS->new();
37          my $data = get( $url );
```

```
        $rss->parse( $data );

        my $channel = $rss->{channel};
        my $image   = $rss->{image};

        print <<"HTML";
        <table cellpadding=1><tr><td bgcolor="#000000">
        <table cellpadding=5>
                <tr><td bgcolor="#aaaaaa" align="center">
HTML

        if( $image->{url} )
                {
                my $img = qq|<img src="$$image{url}" alt="$$channel{title}">|;

                print qq|<a href="$$channel{link}">$img</a><br>\n|.
                }
        else
                {
                print qq|<a href="$$channel{link}">$$channel{title}</a><br>\n|.
                }

        print qq|<font size="-1">$$channel{description}</font>\n|;

        print <<"HTML";
        </td></tr>
        <tr><td bgcolor="#bbbbff" width=200><font size="-1">
HTML

        foreach my $item ( @{ $rss->{items} } )
                {
                print qq|<b>&gt;</b><a href="$$item{link}">$$item{title}</a><br><br>\n|;
                }

        print <<"HTML";
                </font></td></tr>
        </td></tr></table>
        </td></tr></table>
HTML

        close $fh;
        }
```

3.1 Updating RSS feeds automatically

Once I decide which feeds I want to process and how I want to present them, I want to fetch them automatically. I can use crontab to schedule the program to run at certain times (cron comes with unix-like platforms and is available as third-party tools for windows). The frequency that I run this program depends on how often the sites update their feeds. I typically update things hourly, but not on the hour when everyone else is probably updating their feeds. I update at 17 minutes past the hour.

```
17 * * * * /usr/home/comdog/bin/rss2html.pl
```

3.2 Some things I do not cover

The RSS format handles a lot more than what I have shown, but once I have XML::RSS doing the hard work, everything else is easy. Other commonly-used features include search forms linking to the original site, channel descriptions, item descriptions, channel meta-data for caching, fetching, and more.

3.3 Some places offering RSS feeds

None of this parsing magic is much use to anyone unless people offer RSS feeds to parse.

The Perl Review
This journal offers several different RSS feeds to choose from. http://www.theperlreview.com/rss/

"All the Perl that's Practical to Extract and Report"
use.Perl has a feed of its stories - http://use.perl.org/useperl.rss

Recent modules
CPAN Search has a feed of the latest 10 modules

4 Conclusion

With a minimum of effort, I can create or parse RSS files. The XML::RSS module handles the details of different versions for me so I can know as little or as much RSS as I care to know. When I parse an RSS file, I can access its parts through familiar Perl data structures.

5 References

The Perl Review - http://www.theperlreview.com

Perl at a Glance - http://www.theperlreview.com/at_a_glance.shtml

O'Reilly Network RSS DevCenter - http://www.oreillynet.com/rss/

RSS News - http://blogspace.com/rss/

6 About the Author

brian d foy is the publisher of *The Perl Review*.

Delightful Languages: Ruby

Mike Stok, mike@stok.co.uk

Abstract

This is a brief recounting of my initial impressions of and experience with the Ruby programming language and its community. In many ways Ruby strikes the same chord in me that Perl did a decade or more ago. I show Ruby from a Perl perspective.

1 Introduction to Ruby

When I first encountered Perl, I found the language to be a little strange coming from a C background. From time to time I would use Perl to write things I would have written in C or shell, and soon Perl was my tool of choice for many tasks.

Ruby is having a similar effect on me. Sometimes I prototype Perl code in Ruby, sometimes I just use Ruby for the sake of seeing if I arrive at a different solution using a different language. Like Perl, Ruby makes programming fun, but in a different way.

2 Rewriting Soundex

Ruby borrows features from many languages, and one of those is Perl. I can simply translate Perl code into Ruby if I want. I will use the Soundex function as an example, as the algorithm is simple and my Perl implementation will reveal something about my abilities as a programmer.

The Soundex algorithm is a simple hashing of the letters of a word to a four character code which brings similar sounding words to the same code. In 1994 I posted a routine, shown in code listing 1, to comp.lang.perl which shows both the simplicity of the Soundex algorithm and my Perl style at its worst (or best).

Code Listing 1: Original Perl soundex function

```
sub Soundex
{
  local ($_, $f) = shift;

  y;a-z;A-Z;;y;A-Z;;cd;$_ ne q???do{($f)=m;^(.);;s;$f+;;;
  y;AEHIOUWYBFPVCGJKQSXZDTLMNR;00000000111122222222334556;;
  y;;;cs;y;0;;d;s;^;$f;;s;$;0000;;m;(^.{4});;$1;}:q;;;
}
```

This code survives, with a bug fix and some reformatting as the Text::Soundex module in Perl 5.8.0[1], shown in code listing 2. The major difference between the routines is that the newer code makes use of Perl's subroutine call context to decide whether to return a single scalar or a list of scalars. This is a legacy from Perl 4 days when Perl did not have map. These days I would leave it to the routine's user to do the work even if it means a few extra subroutine calls.

Code Listing 2: Current Perl soundex function

```
sub soundex
{
  local (@s, $f, $fc, $_) = @_;

  push @s, '' unless @s;        # handle no args as a single empty string

  foreach (@s)
  {
        $_ = uc $_;
        tr/A-Z//cd;

        if ($_ eq '')
        {
          $_ = $soundex_nocode;
        }
        else
        {
          ($f) = /^(.)/;
          tr/AEHIOUWYBFPVCGJKQSXZDTLMNR/00000000111122222222334556/;
          ($fc) = /^(.)/;
          s/^$fc+//;
          tr///cs;
          tr/0//d;
          $_ = $f . $_ . '000';
          s/^(.{4}).*/$1/;
        }
  }

  wantarray ? @s : shift @s;
}
```

2.1 Ruby Soundex code

I did a simple translation of the Perl code into Ruby[2], shown in code listing 3. As the wantarray is a Perlish idiom I left it out.

[1]Serious Text::Soundex users should grab Mark Mielke's faster version from CPAN. I do not know why the current version has survived so long in the official distribution.

[2]There is already a Soundex module in the Ruby Application Archive. Its author, Michael Neumann, took a more conventional approach to its implementation than mine.

Code Listing 3: Soundex in Ruby

```
def soundex(string, nocode=nil)
        copy = string.upcase.tr '^A-Z', ''
        return nocode if copy.empty?
        first_letter = copy[0, 1]
        copy.tr_s! 'AEHIOUWYBFPVCGJKQSXZDTLMNR',
                   '00000000111122222222334556'
        copy.sub!(/^(.)\1*/, '').gsub!(/0/, '')
        "#{first_letter}#{copy}000"[0 .. 3]
end
```

2.1.1 Perl and Ruby differences

No Semicolons

I did not use any semicolons in code listing 3. Ruby can use semicolons to separate expressions, but most Ruby code uses line ends to indicate the end of an expression or statement. When a line ends in the middle of an expression, Ruby realizes that it will continue on the next line. For example, I can split `x = 2 + 2` over two lines.

```
x = 2 +
    2
```

Ruby sees a single expression as it expects something after the `+`.

If I wanted to put multiple statements on a line then I could use semicolons.

```
x = 2 + 2; y = x + 1
```

No Sigils

Ruby does not use sigils (the leading `$`, `@`, `%` character used in Perl) to indicate the type of a variable. When retrieving an element from a Hash or an Array, the same operator, `[]`, is used. The type of the container I am accessing determines what I should put inside the brackets.

Ruby does use `$`, `@`, and `@@` as prefixes, but I do not cover that in this article.

Named Parameters

Ruby's method definitions let me name parameters and specify default values for optional parameters. The string parameter is required, and the nocode parameter is optional. If I do not specify nocode, then Ruby gives it the value nil.

Ruby checks the arity when I call a method to make sure that I required all parameters. If I try to call soundex with the wrong number of arguments then Ruby raises an ArgumentError.

```
soundex()           # => ArgumentError: wrong number of arguments(0 for 1)
```

No declarations

There are no equivalents of my or local. If I use a variable name then Ruby checks if there is already a variable of that name in scope; if there is, Ruby reuses it, otherwise it creates a variable in the current scope.

String Interpolation Using `#{ ... }`

Inside double quotes Ruby uses `#{ ... }` to interpolate any expression. I used double quotes to concatenate first_character, copy, and the literal `000`, but I can put any expression between the braces and Ruby interpolates the result into the string being generated by the double quotes.

Methods everywhere

All the subroutine calls are methods. Most of the methods, upcase, tr, length, are members of the String class. As in Perl a method is a subroutine called in the context of an object.

Predicate method `empty?`

The `empty?` method ends with a question mark which is part of the method name. In Ruby this conventionally means that the method is a predicate. This convention is not enforced by the interpreter.

In Perl I can say `$string or return $soundex_nocode;` if I know the string cannot contain a solitary 0. Perl's notion of truth means that the empty string or a string containing 0 are considered false. In Ruby the only false values are nil and false, so I have to test string.

I could have used `string.length == 0`, but the name of the `empty?` method expresses my intent more clearly.

Method chaining

`string.upcase.tr '^A-Z', ''` shows how I can chain methods in Ruby. The upcase method takes the value referenced by string, converts it to upper case, then the tr method deletes the non-alphabetic characters. The tr and upcase methods do not affect the value string refers to. Ruby's convention is that a destructive method name ends with a `!`. This is only a convention, so the Ruby interpreter does not enforce it.

Copy data in String

As all data are objects in Ruby, string is a reference to an object. I was careful not to change the value of the original string passed to the subroutine.

When I operate on the data referenced by copy, many of the methods end with a `!`. The tr! method modifies the object it is called on. The tr method always gives you a new string as a result, but tr! returns the modified, original string or nil.

Getting the First Character of a String

Ruby strings are sequences of integers, not arrays of characters.

The first time I tried to get a character from a string in Ruby, I used something like `char = string[0]`. This gave me the ASCII value of the first character of the string. Consulting the section on the String class in *Programming Ruby*, I discovered that I need to use either a Range or a couple of numbers in the brackets to get a substring.

2.2 A Brief Diversion - irb

I use the irb program which comes with Ruby as a test bed for Ruby code, much as I use `perl -de 1` for Perl. irb prints the result of each expression it evaluates, as in code listing 4. The `irb(main):001:0>` is the irb prompt. When I enter an expression, the result irb prints before the next prompt.

Code Listing 4: irb session

```
[mike@ratdog tmp]$ irb
irb(main):001:0> string = "Mike"
"Mike"
irb(main):002:0> string.class
String
irb(main):003:0> string[0]
77
irb(main):004:0> string[0].class
Fixnum
irb(main):005:0> string[0,1].class
String
irb(main):006:0> string[0,1]
"M"
```

2.3 Making soundex More Rubyesque

The Ruby soundex code presented in code listing 3 is fragile. When I use it on strings, all is well, but should I accidentally use it on a variable containing a number, as in code listing 5, something else happens. Perl automatically morphs the contents of scalars between numbers and strings, but Ruby expects you to be explicit about this. If an object does not does not respond to a method, Ruby complains.

Code Listing 5: Passing Ruby's soundex a number

```
irb(main):018:0> soundex(2)
NoMethodError: undefined method ‘upcase’ for 2:Fixnum
                from (irb):8:in ‘soundex’
                from (irb):18
                from :0
```

The things I can do to mitigate this include:

Argument Checking
I can test the class of the argument using the class method available to all Ruby objects.

Turn the Argument into a String
I can use the to_s method which turns Ruby objects into Strings.

Make soundex a String Method
I can put soundex into the String class and make it look more like a builtin.

I prefer to make soundex act more like a builtin String method. Ruby allows me to add methods to classes at any time. I can add soundex to the String class so that it is available to Strings and classes derived from them for the duration of the program.

When I use soundex as a String method call, the thing being encoded is available through a variable self or I can use implicitly as the default object. In the code listing 6 there is an implied self in the `copy = upcase.tr '^A-Z', ''` which I could write as the equivalent `copy = self.upcase.tr '^A-Z', ''`.

Code Listing 6: Adding soundex to String

```
class String
        def soundex(nocode=nil)
                copy = upcase.tr '^A-Z', ''
                return nocode if copy.empty?
                first_letter = copy[0, 1]
                copy.tr_s! 'AEHIOUWYBFPVCGJKQSXZDTLMNR',
                           '00000000111122222222334556'
                copy.sub!(/^(.)\1*/, '').gsub!(/0/, '')
                "#{first_letter}#{copy}000"[0 .. 3]
        end
end
```

I saved code listing 6 in soundex.rb, and running irb in the same directory as soundex.rb I could make soundex available to all Strings and objects in subclasses of String. In code listing 7, I ran an irb session to try it. The soundex method is not available until I load the file containing String#soundex, soundex.rb, with require.

Code Listing 7: Importing the soundex method

```
irb(main):001:0> str = "Mike"
"Mike"
irb(main):002:0> str.soundex
NoMethodError: undefined method `soundex' for "Mike":String
                from (irb):2
irb(main):003:0> require 'soundex'
true
irb(main):004:0> str.soundex
"M200"
```

In code listing 8 I create a Surname subclass of String and check that soundex is available. The < in the class line means that Surname is a subclass of String. In the last couple of lines I show that surname really is a Surname object, and that I can tell that surname's class is derived from String. In Perl I would use something like `$s->isa('String')` to do this.

Code Listing 8: Creating the Surname subclass

```
irb(main):005:0> class Surname < String
irb(main):006:1> end
nil
irb(main):007:0> surname = Surname.new('Stok')
"Stok"
irb(main):008:0> surname.soundex
"S320"
irb(main):009:0> surname.class
Surname
irb(main):010:0> surname.kind_of? String
true
```

2.4 Testing the Code

The soundex code seems to work, but I do not feel happy until I have a basic set of tests I can run to make sure that innocuous changes do not break things.

Ruby's equivalent to CPAN is the Ruby Application Archive (RAA). I use Nathaniel Talbott's Test::Unit package which makes writing and running test cases a breeze.

To use Test::Unit, I just have to put all my test cases in a class derived from Test::Unit::TestCase, and Test::Unit will find all the methods and run named test_* in that class. Test::Unit has a number of assertion methods I can use, but my test cases are so simple that they only use a few.

Ruby has the __FILE__ token, so I can add a line to my module to see if the module (soundex.rb) is the program being run or whether it is being included from another file. If it is being included Ruby should not run the tests, and if I run the file directly, it should run the tests.

To add unit tests to the code presented above I add them to the bottom of the file, as in code listing 9.

Code Listing 9: Testing soundex

```
if __FILE__ == $0
        require 'test/unit'

        class TC_Soundex < Test::Unit::TestCase
                def test_knuth
                        [ %w(Euler       Ellery    E460),
                          %w(Gauss       Ghosh     G200),
                          %w(Hilbert     Heilbron  H416),
                          %w(Knuth       Kant      K530),
                          %w(Lloyd       Ladd      L300),
                          %w(Lukasiewicz Lissajous L222),
                        ].each do |test|
                                assert_equal(test[0].soundex, test[-1])
                                assert_equal(test[0].soundex, test[1].soundex)
                        end
                end

                def test_empty
                        assert_nil(''.soundex)
                end

                def test_non_alpha
                        assert_nil('2+2=4'.soundex)
                end

                def test_mike
                        assert_equal('Mike'.soundex, 'M200')
                        assert_equal('Stok'.soundex, 'S320')
                end

                def test_czarkowska
                        # in the old perl version this was a bug which caused a
                        # discrepancy between Oracle's soundex and mine.  Spotted
                        # by Rich Pinder

                        assert_equal('CZARKOWSKA'.soundex, 'C622')
                end

                def test_nocode
                        assert_equal(''.soundex('?'), '?')
                end
        end
end
```

I run the tests by executing the module directly.

```
[mike@ratdog ruby-soundex]$ ruby soundex.rb
Loaded suite soundex
Started
.......
Finished in 0.041179 seconds.
6 tests, 18 assertions, 0 failures, 0 errors
```

2.5 Packaging the Code

When I get a Perl module from CPAN, I know that I can install it with the familiar `perl Makefile.PL`, `make`, `make test` and `make install` ritual.

The Ruby module world has not yet settled into a predictable pattern for module installation. If I want to use code from a library file, then the file has to be in one of the directories mentioned in the `$:` variable which is like Perl's @INC (Ruby does not have built-in formats).

3 Impressions of Ruby

I really like Ruby. I thought I would like it when I saw it first more than a year ago, and I still like it. I still use Perl, and I think that Ruby has improved my Perl.

I find Ruby's clean class and method definitions make me much more inclined to make new classes for different types of object in my code. I can use various helper modules like base in Perl, but I prefer the look of Ruby.

I try to adopt the Extreme Programming "test first" strategy when developing Ruby code. This means that I am less likely to get caught out by the lack of explicit scoping commands. I like the control that my gives me in Perl. While giving me more control, my does clutter up the code and lets me get away with long rambling subroutines. Ruby encourages me to write small routines and think about minimizing the scope of variables.

The abundance of unit tests provided with most of the code on the Ruby Application Archive makes me much more comfortable when I am trying to fix flaws in code. When I needed to fix a couple of problems in Sean Russell's amazing Ruby Electric XML (REXML) package, the unit tests allowed me to see how much damage I was doing.

I have avoided going over areas which are well covered in other articles which introduce Ruby. I found the *Doctor Dobb's* article by Dave Thomas and Andy Hunt a good introduction to Ruby's features. I found that Ruby has enough features which are different enough from those in Perl that it made me think differently about problems; if I had to pick one to begin with then it would be the yield method.

3.1 A yield Digression

I can use yield to build generators and iterators. I can do this in Perl and Python too, but Ruby's libraries make so much use of iterators that they do not seem at all unusual.

The Ruby Enumerable module class uses yield in the `select` which selects items from a collection by calling a user specified block of code on each element of the collection. This is much like Perl's `grep BLOCK LIST`, where grep calls BLOCK on each element of LIST. I use this to select all the elements smaller than a pivot from an array in Ruby.

```
lesser  = list.select { |e| e <  pivot }
```

The block is called on each element of C¡list¿, if the element is less than the pivot then that element is included in the list which is returned. In Perl I perform the selection using this code:

```
my @lesser = grep { $_ < $pivot} @list;
```

In Perl the code block comes between grep and LIST, and in the Ruby the block comes at the end. I prefer the Ruby style, especially if I need to split the code across multiple lines.

When I want to allow the user to pass a block of code to a method I just call yield with the argments I want to pass into the block. Python recently acquired yield, so the simple Ruby transliteration of Paul Prescod's fib_gen2 routine from *The Perl Review (0,2)* might look like code listing 10. The fib_gen2 method yields control to a user supplied block of code each time around the count.times loop.

Code Listing 10: Ruby fibonacci number generator

```
def fib_gen2(count)
        this_number, next_number = 1, 1
        count.times do
                yield this_number, next_number
                this_number, next_number = next_number, this_number + next_number
        end
end

fib_gen2(100) { |n1, n2|
        puts "#{n1} #{n2}"
}
```

4 Ruby Resources

The Ruby community is active and varied. There are many local Ruby User Groups, and there is an annual Ruby Conference.

One of the things which attracted me to Ruby was its community. The spirit reminds me of the Perl community in the early 1990s.

There are many Ruby resources on the net. Over the past couple of years Ruby resources have started to be available in more languages than Japanese and English. There are links to the entire text of *Programming Ruby*.

The comp.lang.ruby Newsgroup

Ruby's author and many active contributors participate in the news group. I find the traffic moderate and the signal-to-noise ratio high.

#ruby-lang on irc.openprojects.net

There is a Ruby IRC channel. I don't use IRC.

Ruby Language and Ruby Garden Web Sites

The main Ruby language web site is at http://www.ruby-lang.org . This site contains pretty much everything you need to explore Ruby and get started. It has links to other introductory articles.

The community web site for Ruby is http://www.rubygarden.com . This has all kinds of useful resources including a Ruby wiki and a summary of the past week's activity on the newsgroup.

Ruby Application Archive (RAA)

The Ruby Application Archive is Ruby's equivalent to CPAN. There is a link to it on the Ruby language web site mentioned above.

If I wanted to use Perl-style formats then a search of the RAA would get me to Paul Rubel's FormatR package (billed as "just like Perl's plus some and the ability to go in reverse.")

Ruby Books

The most popular English language book about Ruby seems to be *Programming Ruby* by David Thomas and Andrew Hunt. This book had the same effect on my appreciation of Ruby as the first edition of *Programming Perl* had on my interest in Perl. My copy has become quite dog-eared—always a good sign that I find a book useful.

Programming Ruby is available on-line at http://www.rubycentral.com/book/index.html . This is a perfect complement to the print version, and the introduction to Ruby is a good read.

Hal Fulton's *The Ruby Way* is an excellent source of all kinds of techniques you can use in Ruby. It is task-oriented and includes an appendix on coming to Ruby from Perl.

5 Conclusion

I like Ruby because it seems to have struck the right balance between simplicity and power. The language is different enough from Perl to make me think about things in a fresh light. The Ruby community is active and helpful. The texture of Ruby encourages me to write code which seems to make sense months later.

I feel like the time I have spent with Ruby has been as rewarding as the time I spent with Perl. Admittedly the Ruby Application Archive is not as big as CPAN, and some of Ruby's features might appear in Perl 6, but I still think it is worth giving Ruby a try.

6 References

The Ruby Language - http://www.ruby-lang.org/

"Programming in Ruby", Dave Thomas and Andy Hunt, Dr. Dobbs, January 2001,http://www.ddj.com/documents/s=871/ddj0101b/0101b.htm

http://www.germane-software.com/software/rexml/ - the REXML module. The tutorial for this package might give you a better feel for Ruby's character.

One way of insinuating Ruby code into a Perl world is the devious use of Brian Ingerson's Inline::Ruby module from CPAN.

7 About the Author

I work for Exegenix in Toronto, writing Perl to transform documents into other documents. I enjoy life with my wife and daughter, Perl and Ruby coding, hot air ballooning, beer and chocolate (even white chocolate, as long as it's a Toblerone).

Who's Doing What? Analyzing Ethernet LAN Traffic

by Paul Barry, paul.barry@itcarlow.ie

Abstract

A small collection of Perl modules provides the basic building blocks for the creation of a Perl-based Ethernet network analyzer. I present a network analyzer that captures local area network (LAN) traffic destined for the local domain name system (DNS) server and logs information on which IP addresses request which IP name resolutions. The developed program can form the basis of any custom Ethernet LAN analyzer.

1 Introduction

Plenty of tools can capture and analyze network traffic. On Ethernet local area networks (LAN), EtherPeek and Surveyor are heavy-hitting commercial offerings, whereas on Unix platforms, there is tcpdump and Ethereal. These are excellent tools that provide a large, and sometimes intimidating, set of features.

At times, though, I need something else. Perhaps I need to analyze a new or custom protocol not allowed for by existing tools, or I need to process each captured chunk of network traffic. In these cases, I need a programming language that can talk to the LAN.

I could always use C. On Unix platforms, the libpcap library provides an Ethernet packet-capturing programming interface written in C and is used by both tcpdump and Ethereal, but dropping down to C takes too much time when I am in a hurry.

2 Perl and CPAN to the rescue!

The Comprehensive Perl Archive Network (CPAN) has a number of module interfaces to libpcap. My favourite is Tim Potter's Net::Pcap which is a blow-for-blow mapping of the libpcap interface into its Perl equivalent.

To make the most of Net::Pcap I have to study the pcap documentation and this is very tough going. Thankfully, Tim provides a companion module, Net::PcapUtils, which abstracts all the nitty-gritty details of Net::Pcap into a very small collection of functions and methods. When combined with his NetPacket module which can decode and encode a growing collection of network packets, Net::PcapUtils forms the basis of an network analyzer.

The NetPacket module can decode raw Ethernet frames, Internet Protocol (IP) datagrams, Unix Domain Protocol (UDP) datagrams and Transmission Control Protocol (TCP) segments, which corresponds to the three lower levels of the TCP/IP Reference Model: the Host-to-Network, Network , and Transport Layers, respectively. NetPacket does not handle the final, topmost layer, the Application Layer.

3 Who's Doing What?

The Who's Doing What (wdw) program in code listing 1 captures and processes details of the IP addresses which are resolving IP names against the local DNS server. In addition to displaying the results on screen as they happen, it logs each request to a file.

On lines 5 and 6, I define two constants, DNS_PORT and HOWMANY. The DNS_PORT is the standard protocol port number for DNS, and HOWMANY is the default number of packets to capture.

On line 14, I declare and define a global variable, $num_processed, which I use to record the number of packets I process.

My first programming language was Pascal which explains the definition of my subroutines before I actually use them. I explain these later.

On line 59, I set the number of packets to process. If I do not provide a command line argument, I use the value of HOWMANY. Immediately upon setting $count, I remember its initial value in $rem_count because my while loop, starting on line 76, changes the value of $count.

On line 61, I call the Net::PcapUtils::open function which places the Ethernet card into promiscuous mode for packet capturing. The two parameters to open specify the protocol filter—DNS uses UDP—and the maximum amount of data to capture for each packet—1500 bytes is the maximum payload size on Ethernet networks. If open succeeds, $pkt_descriptor is reference to a valid packet capture descriptor. If it fails it returns an error message which I check for this on line 65. On error I display an appropriate error message and then exit.

On line 71, I open the log file in append mode and on line 74 I timestamp it at the beginning of the run. On line 79, I send the output from got_a_packet to this file.

The while loop on lines 76 to 81 is the guts of the program. With each iteration, I call the Net::PcapUtils::next subroutine with my packet capture descriptor in $pkt_descriptor. This subroutine waits for a UDP packet then returns two values—a scalar which represents the raw Ethernet packet, which I store in $packet, and a hash, which I do not need. I pass the raw Ethernet packet to got_a_packet, which also takes the log filehandle. The while loop iterates $count times.

On lines 17 and 18, in got_a_packet, I assign the two parameters to $handle and $packet. I will append the results to the log file using $handle and use the contents of $packet to create a NetPacket::IP object. On lines 20 and 21, the NetPacket::Ethernet::eth_strip subroutine removes the Ethernet frame from $packet and returns the IP datagram, and NetPacket::IP::decode extracts the IP data portion. The UDP datagram is in the data portion of the IP datagram.

Once I have the UDP object, on line 25 I check if the destination port is the port stored in DNS_PORT. If it is, on line 27 I assign the data portion of the UDP datagram to $dns_packet.

The NetPacket modules knows nothing of the protocols running at the Application Layer, the level of DNS. On line 28, I use the Net::DNS::Packet module to decode the data portion from the IP packet, then on line 29, I extract the DNS queries from that and store them in @questions.

One line 31, I iterate over @questions and process each DNS query. On line 33, I get the current query in stringified form, and on line 35 I skip queries that match `in-addr.arpa` since those relate to IP addresses, not names.

Lines 39 to 44 outputs the source and destination IP addresses together with the IP name to the screen and log-file. On line 46, I increment $num_processed since I have processed the right sort of query.

On lines 52 to 57, the display_results subroutine prints a summary message that shows the number of packets processed, in $num_processed, and the number of packets actually processed, in $outof.

Code Listing 1: The Who's Doing What program

```
 1 #!/usr/bin/perl -w
 2 use 5.6.0; # Change to 5.006_000 if using Perl 5.8.0.
 3 use strict;
 4
 5 use constant DNS_PORT     => 53;
 6 use constant HOWMANY      => 100;
 7
 8 use Net::DNS::Packet;
 9 use Net::PcapUtils;
10 use NetPacket::Ethernet qw( :strip );
11 use NetPacket::IP;
12 use NetPacket::UDP;
13
14 our $num_processed = 0;
15
16 sub got_a_packet {
17   my $handle = shift;
18   my $packet = shift;
19
20   my $ip_datagram = NetPacket::IP->decode(
21                                                 NetPacket::Ethernet::eth_strip( $packet ) );
22
23   my $udp_datagram = NetPacket::UDP->decode( $ip_datagram->{data} );
24
25   if ( $udp_datagram->{dest_port} == DNS_PORT )
26   {
27           my $dns_packet = $udp_datagram->{data};
28           my $dns_decode = Net::DNS::Packet->new( \$dns_packet );
29           my @questions = $dns_decode->question;
30
31           foreach my $q ( @questions )
32           {
33                   my $question = $q->string;
34
35                   unless ( $question =~ /in-addr\.arpa/ )
36                   {
37                           $question =~ /^(.+)\tIN/;
38
39                           print "$ip_datagram->{src_ip} -> ";
40                           print "$ip_datagram->{dest_ip}: ";
41                           print "$1\n";
42                           print $handle "$ip_datagram->{src_ip} -> ";
43                           print $handle "$ip_datagram->{dest_ip}: ";
44                           print $handle "$1\n";
45
46                           $num_processed++;
47                   }
48           }
```

```
  }
}

sub display_results {
  my $outof = shift;

  print "\nProcessed $num_processed (out of $outof) ";
  print "UDP datagrams carrying DNS.\n\n";
}

my $count = shift || HOWMANY;
my $rem_count = $count;
my $pkt_descriptor = Net::PcapUtils::open(
                                           FILTER => 'udp',
                                           SNAPLEN => 1500 );

if ( !ref( $pkt_descriptor ) )
{
  warn "Net::PcapUtils::open returned: $pkt_descriptor\n";
  exit;
}

open WDW_FILE, ">>wdw_log.txt"
  or die "Could not append to wdw_log.txt: $!\n";

print WDW_FILE "\n", scalar localtime, " - wdw BEGIN run.\n\n";

while ( $count )
{
  my ( $packet, %header ) = Net::PcapUtils::next( $pkt_descriptor );
  got_a_packet( *WDW_FILE, $packet );
  $count--;
}

print WDW_FILE "\n", scalar localtime, " - wdw END run.\n";
close WDW_FILE;

display_results( $rem_count );
```

4 Running wdw

I login as root to run the wdw program because I need to put the Ethernet network interface card (NIC) into promiscuous mode, which is a restricted activity that only root can do.

The default number of packets to capture is 100. I supply a single, positive integer argument to wdw on the command line if I want to capture a different number.

The program output shows up on standard output and in the log file. I ran the program with fictitious IP addresses I already set up to create the sample in code listing 2. The address 10.0.0.5 provides DNS to my fictitous local network.

Code Listing 2: wdw output

```
 1  10.0.0.65 -> 10.0.0.5: www.theprelreview.com.
 2  10.0.0.65 -> 10.0.0.5: www.theprelreview.com.itcarlow.ie.
 3  10.0.0.65 -> 10.0.0.5: www.theperlreview.com.
 4  10.0.0.200 -> 10.0.0.5: www.perl.com.
 5  10.0.0.200 -> 10.0.0.5: search.cpan.org.
 6  10.0.0.39 -> 10.0.0.5: www.linuxjournal.com.
 7  10.0.0.88 -> 10.0.0.5: www.apple.com.
 8  10.0.0.118 -> 10.0.0.5: glasnost.itcarlow.ie.
 9  10.0.0.118 -> 10.0.0.5: www.wileyeurope.com.
10  10.0.0.118 -> 10.0.0.5: www.wiley.com.
11  10.0.0.65 -> 10.0.0.5: www.amazon.com.
12
13  Processed 11 (out of 100) UDP datagrams carrying DNS.
```

On line 1 of code listing 2, the misspelling of "prel" instead of "perl", causes the lookup to fail. The client program, upon failing to lookup www.theprelreview.com, tried to resolve the name as if it were a host on the local internet domain, itcarlow.ie. This accounts for the rather long hostname on line 2.

5 Conclusion

Any half-decent network manager (or sysadmin) will tell me the information logged by wdw is more than likely logged by my DNS server. It is easy for most Perl programmers to write a quick program to extract the data items of interest from the DNS log. However, wdw displays the data as it happens, whereas instead of after-the-fact. I can also use this as the basis of more complicated real-time analyzers.

6 References

The pcap(3) manual page.

Module documentation - NetPacket, Net::Pcap, Net::PcapUtils, and Net::DNS.

Tim Potter's modules - http://search.cpan.org/author/TIMPOTTER.

Michael Fuhr's Net::DNS website - http://www.net-dns.org.

Internet RFCs: 1034 and 1035 for the original DNS standards.

Programming the Network with Perl, chapter 2, by Paul Barry. John Wiley & Sons Limited, 2002, ISBN 0-471-48670-1.

libpcap library - http://www.tcpdump.org

wdw source code - http://glasnost.itcarlow.ie/˜barryp/wdw.tar.gz

7 About the Author

Paul Barry (paul.barry@itcarlow.ie) lectures in Computer Networking and Systems Administration at The Institute of Technology, Carlow in Ireland. Paul is the author of *Programming the Network with Perl* (Wiley, 2002). He occasionally writes for *Linux Journal* magazine and web site.

Book Reviews

Extending and Embedding Perl

Tim Jenness and Simon Cozens, Manning Press, 1-930110-82-0, August 2002; 312 pages
reviewed by Andy Lester

The great thing about the Perl book business these days is that publishers seem to have cycled their way through all the general books and are focusing on narrower topics. These books are not for everyone, but they're authoritative, and are just the book you need for certain tasks. Recent examples include "Programming The Network With Perl", "Writing Perl Modules For CPAN" and the new "Extending and Embedding Perl".

Everything related to Perl internals is found here: the Perl internal API, XS, SWIG, embedding Perl in C, and C in Perl. The book is organized well, and is fine as a reference work. Where the book falls flat is as an introduction.

The presentation is more as a shop manual than a tutorial. It's almost as if the book was written for someone already familiar with some of these concepts, rather than to the experienced Perl programmer looking to work outside the language itself. I would also like to have seen more practical examples geared toward the real user. One example talks about embedding FORTRAN code into a Perl program, which seems like a fairly unlikely Perl extension for most readers. Another example shows how to replace the configuration handling for the mutt mail reader without even addressing why anyone would want to do such a thing.

Also, experienced C programmers will have to skip 45 pages of an overview on C. I was surprised to see it until I realized that here in the 21st Century there may actually be people who do not know C. Am I showing my age? Of course, any book that delves into Perl internals is going to have all sorts of gory C details.

Overall, there's no life or humor in the writing, and it is got the zest of an economics textbook. As a reference, that's fine. For an introduction to this arcane world, something needs to be there to help pull the reader through.

For the reader who needs something on paper, or needs a solid reference, this book is authoritative. If you are looking for something to get you started in the arcane world of Perl internals, I'm afraid you'll be as disappointed as I was.

Books on Ruby

reviewed by brian d foy

I started learning Ruby so I could work on Mike Stok's article in this issue. As with any new thing I learned, I surveyed the books available.

Ruby In A Nutshell, by Yukihiro "Matz" Matsumoto (O'Reilly & Associates, 2002, 204 pages, 0-596-00214-9) was the Ruby book I thought would be most useful, but the class reference is organized by type rather than alphabetically which makes me do a lot of work to figure out where I need to look. I constantly had to use the index, which annoyed me. Other than that, the book is up to O'Reilly's usual standards, although I do not recommend it.

Programming Ruby, by Dave Thomas and Andrew Hunt (Addison-Wesley, 2001, 564 pages, 0-201-71089-7) is a delight to read, even if the order of topics is a bit strange. If you already know a bit about programming and have worked with another object-oriented language, even if that was just Perl, you should have an easy time learning Ruby from this book. The class reference at the end of the book is in alphabetic order, so you do not need *Ruby In A Nutshell* if you have this book.

The Ruby Way, by Hal Fulton (Sams, 2002, 579 pages, 0-672-32083-5) is mostly a Ruby cookbook. The first half of the book deals with very simple recipes for common tasks, and the latter half deals with more complex and less frequently encountered problems. The author put a lot of code into this book. The introduction by Matz, Ruby's creator, and Hal's preface provide a good introduction to the Ruby mindset. You should read even if you do not need the book.

The Ruby Developer's Guide by Robert Feldt and Lyle Johnson (Syngress, 2002, 693 pages, 1-928994-64-4) is a task-oriented book. It covers GUI toolkits, databases, XML, and web services along with bits of Ruby wisdom and developer tools. If you do not already know Ruby, save this book for later.