COLUMNS

Practical Perl Tools: Hither and Yon

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

Today's column will be all about hither and yon. Actually, this column will be more about moving files from hither to yon or perhaps yon to hither (not entirely sure which; pity William Safire didn't write more Perl modules while he was alive). If you've ever had to move data around, this will be the column for you.

Built-In Functions and Core Modules for File Copying/Moving

Let's start close to home with the function and modules that ship with Perl. Perl has a rename() function should you want to change the name of a file, perhaps moving it in the process to a different place in the current file system. To copy data, there's always code like this:

```
open my $OLDFILE, '<', 'oldfilename' or
   die "Can't open oldfilename for reading:$!\n";
open my $NEWCOPY, '>', 'newfilename' or
   die "Can't open newfilename for writing:$!\n";
# binmode() makes sure no end-of-line translation is
# done on the OSes that make a distinction
# between binary and non-binary files
binmode $OLDFILE;
binmode $NEWFILE;
while (<$OLDFILE>) { print {$NEWCOPY} $_; }
close $NEWCOPY;
```

but that is so gauche. Better would be to use the File::Copy module, helpfully shipped with Perl since 5.002. Barring one twist, you'd be hard-pressed to find a more straightforward syntax:

```
use File::Copy;
copy('oldfilename','newfilename') or die "Copy failed: $!";
```

What's the twist? Sometimes you really want to write this instead:

```
use File::Copy 'cp';
cp('oldfilename','newfilename') or die "Copy failed: $!";
```

In the first sample, copy() creates a copy of the file and gives it the default permissions (e.g., from the active umask); in the second, cp() attempts to preserve the source file's permissions when creating the copy.

The File::Copy module also has a move() function that attempts to move the file in the most efficient way possible. If it can just rename() the file it will; otherwise it will attempt to copy the file and delete the source file.

One quick caveat about this and other similar modules: by default, it won't copy attributes that are specific to a particular file system/OS. For example, if you are trying to copy a file on NTFS, File::Copy won't copy the ACLs on the file by default. The module provides a syscopy() function that can address this concern under certain conditions when the right external modules are present. See the documentation for more information if you think you will need to care about this sort of thing.

Extending File Copy Functionality

Once we have basic file copies down, we can start extending this idea using some other non-core modules. Copying a single file is useful, but perhaps being able to copy an entire tree of files/directories is even better. File::Copy::Recursive is made to do just that. It offers a bit more control than you probably thought you needed, by providing separate functions for recursively copying either all of the files in a directory structure, all of the directories, or both. The syntax is equally easy to use:

```
use File::Copy::Recursive qw(fcopy dircopy rcopy fmove dirmove rmove);

fcopy($orig,$new) or die $!;   # copy files
dircopy($orig,$new) or die $!; # copy directories
rcopy($orig,$new) or die $!;   # copy either

fmove($orig,$new) or die $!;
dirmove($orig,$new) or die $!;
rmove($orig,$new) or die $!;
```

There are a number of option variables you can set to modify the behavior of these functions; be sure to check out the documentation.

Perhaps a more interesting module is File::Copy::Vigilant, which describes itself as "A module for when your files absolutely, positively have to get there." File::Copy::Vigilant code looks awfully similar:

```
use File::Copy::Vigilant;

copy_vigilant( $source, $destination );
move_vigilant( $source, $destination );
```

but what it does under the hood is even cooler. Before a copy or move takes place, the module computes the MD5 digest for the source file. It similarly computes it for the destination file after the operation takes place. If these two digests don't match up, File::Copy::Vigilant will retry the operation a configurable number of times. If you think an MD5 digest comparison is problematic for some reason (e.g., too computationally intensive), you can change it so that it tests file sizes or even does a byte-for-byte comparison of both files.

Bring on the Network Protocols: FTP

So far we've only discussed moving files around in cases where the source and the destination were both directly accessible as a file system from the local machine. But in this space age of plastics and Intergoogles, clearly that isn't always going to be the case. We need a way to automate the transfer of files between machines that are more loosely connected.

The first protocol for doing this that comes to mind is FTP. After all, it is the File Transfer Protocol. There are a number of modules to work with FTP, but we're not going to spend much time on the subject, largely because my sense is that FTP usage has waned tremendously over the years. I'm not going to get all hyperbolic and suggest it is a dying protocol, but the number of times I've fired up an FTP client in the last year can be counted on one hand, give or take a margin of error of a finger or two. Let's look at the basics and then I'll mention a few modules that could come in handy if for some reason you find yourself having to do more FTP transfers than you expect.

The seminal FTP-based module is Net::FTP. Net::FTP is shipped with Perl as part of the equally seminal libnet package. Here's the example from the documentation:

```
use Net::FTP;

$ftp = Net::FTP->new("some.host.name", Debug => 0)
  or die "Cannot connect to some.host.name: $@";

$ftp->login("anonymous",'-anonymous@')
  or die "Cannot login ", $ftp->message;

$ftp->cwd("/pub")
  or die "Cannot change working directory ", $ftp->message;

$ftp->get("that.file")
  or die "get failed ", $ftp->message;

$ftp->quit;
```

The code creates a new $ftp object using the FTP server name as an argument. With this object, we can log in to the server (in this case anonymously), change the working directory, get a file (and put one, though that's not demonstrated above) and then log out.

If we wanted to build on this basic functionality, we could use modules like:

Net::FTP::Recursive—to put and get entire directory structures
Net::FTP::AutoReconnect—to reconnect on failure
Net::FTP::Throttle—to limit the bandwidth used during transfers
Net::FTP::Find—to walk directory trees on a remote FTP server

But as I mentioned, FTP is a bit passé. All the cool kids are most likely using something else. Let's look at two of the alternatives.

Secure Shell (SSH/SCP/SFTP)

Moving files around in a secure fashion these days is commonly done either via an SSH-based method or a SSL/TLS-based method. The latter will be discussed in a future column, so let's focus for the moment on SSH-based methods. The two SSH-based methods that are almost always used are SCP and SFTP. If I had my druthers I'd just show you how they are used from Perl without even bothering to mention anything about the details of the SSH implementations they are based on. But, alas, in order to understand which ones are appropriate under which circumstances, we're going to have to have a little chat about the types of Perl SSH modules.

The trickiest thing about using SSH-based modules in Perl is deciding which one of the several branches to use. Let's meet the three contestants:

1. Perl-ish implementations—The best example of this type is Net::SSH::Perl, which aims to provide a complete SSH1 and SSH2 implementation in Perl. I say "Perl-ish" because in order to handle some of the more computationally intensive work, Net::SSH::Perl depends on other modules that either call external libraries or are partially implemented in C.
2. Thin veneer around the libssh2 (www.libssh2.org) library—libssh2 describes itself as a "client-side C library implementing the SSH2 protocol." The module Net::SSH2 lets you use all of its power from Perl.
3. Wrappers around existing SSH binaries—Modules like Net::OpenSSH provide pleasant Perl interfaces to what essentially amounts to "shelling-out" to call the appropriate SSH client binary. (This isn't necessarily as bad as it initially sounds, as we'll see in a moment.)

So, how do you choose between them? It really depends on what is most important to you. The first option is good if you don't want to be concerned about installing SSH client binaries or the libssh2 module. The minus of that option is it has quite a few dependencies, some of which used to be a bear to get built and installed. I haven't tried installing it in a while, though, so perhaps this has improved some over time.

The second option is good if you like the notion that there is a single library that does all of the heavy lifting connected to relatively straightforward Perl code to use it. Its drawback is that you have to trust that the library is solid, well tested, and performant enough for your comfort level. The libssh2 team has made excellent progress over the past year or so, so perhaps this is less of a concern.

Finally, the last option is good if you approve of the idea that the Perl code will be depending on (e.g., in the case of Net::OpenSSH) binaries that have undergone a tremendous amount of scrutiny for security issues and are in active use by countless numbers of people every day as a crucial part of their work. The other plus of these modules is that they reuse any existing SSH-related configuration (like config files and SSH keys) you already have in place. As I alluded to above, its minus relates to having to spin up a separate secondary process each time work needs to be done. Net::OpenSSH tries to mitigate that performance hit to a certain extent by making use of OpenSSH-specific multiplexing. This lets you run multiple ssh commands over the same session without having to spin up a new session for each. That can be a big efficiency win in certain cases.

Now that we've discussed the plumbing underneath the SSH file transfer modules, let's talk about what's available in that space. If you want to use SCP, the first file transfer protocol available with SSH, there are two options which both come out of category #3 above: Net::OpenSSH and Net::SCP. Here's an example of using the former to transfer a file:

```
use Net::OpenSSH;

my $ssh = Net::OpenSSH->new($host);
$ssh->error and
  die "Couldn't establish SSH connection: ". $ssh->error;

$ssh->scp_put("localfile", "/remotedir")
  or die "scp failed: " . $ssh->error;
```

With the introduction of SSH version 2 came SFTP, an additional file transfer protocol that mimicked FTP in its interface. There are a greater number of

Perl SFTP options than there are SCP. These include Net::SFTP (built on top of Net::SSH::Perl), Net::OpenSSH (yup, it does both), Net::SSH2::SFTP (built on Net::SSH2), and Net::SFTP::Foreign (which calls the already installed ssh binaries). At the moment, my module of choice in this space is Net::SFTP::Foreign, because it is the most comprehensive of the bunch and relies on binaries I already trust. Here's some code that performs the same task as the previous sample:

```
use Net::SFTP::Foreign;
my $sftp = Net::SFTP::Foreign->new($host);
$sftp->error
  and die "Unable to establish SFTP connection: " . $sftp->error;

$sftp->put("localfile", "/remotedir")
  or die "put failed: " . $sftp->error;
```

Net::SFTP::Foreign also has cool methods like find() to search the remote file system, mget()/mput() to transfer multiple files based on wildcards, and readline() to read a line at a time from a remote file.

As an aside before we move on, there are a few spiffy SSH-based modules we won't be able to cover here, like SSH::Batch and SSH::OpenSSH::Parallel, both for running commands on multiple hosts in parallel. Check them out on CPAN if you are interested.

Rsync

I've often said that I think one of Andrew Tridgell's greatest gifts to sysadmins (and to others) is rsync, both the tool and the algorithm. (Note: I'm cheating a bit by including this in a list of protocols, but bear with me.) At LISA '10, I asked my classes if anyone in the room hadn't yet heard of rsync. I was pleased to see no hands raised. If you happened to be hanging out with Plato in his cave and haven't heard of rsync, the one-sentence summary might be something like "rsync is a very efficient file transfer utility based on an algorithm that lets it send just differences between the source and the destination over the wire." It can use other transports (e.g., SSH) for security if desired. More info can be found at rsync.samba.org.

The types of Perl rsync modules pretty closely mirror those we talked about for SSH. The one difference is there really is no good option for #2 above (i.e., using a Perl wrapper around a library) because librsync itself (sourceforge.net/projects/librsync/) is no longer actively maintained and hasn't seen a new release in over four years. What we do have is an all-Perl implementation called File::RsyncP and a wrapper around the standard rsync client called File::Rsync. I think you'd have to have a good and probably specialized reason to use the pure Perl version (e.g., it was originally written to be used as part of BackupPC), so here's an example from the File::Rsync docs:

```
use File::Rsync;
$obj = File::Rsync->new( {  'archive'    => 1,
                            'compress'   => 1,
                            'rsh'        => '/usr/local/bin/ssh',
                            'rsync-path' => '/usr/local/bin/rsync' } );

$obj->exec( { src => 'localdir', dest => 'rhost:remdir' } )
  or warn "rsync failed\n";
```

If it looks to you like we're basically just writing out the standard options to the rsync command using Perl syntax, that's exactly right.

What About the Rest?

Oh dearie me. We're about to run out of space and there are still tons of things we haven't talked about. For example, you probably heard of this newfangled protocol for moving data around called HTTP. I predict it will be pretty big by the time you read this. There's probably going to be a whole other column in our future on just that protocol.

I don't know if this will mollify you until then, but let me end this column by handing you the keys to your very own nuclear device. If you want to have some fun, in a Dr. Strangelove kind of way, you might want to check out Net::CascadeCopy. In this column we've talked about moving files from one place to another, but largely in a "onesies-twosies" approach. What if you want to move data to a kerjillion servers? Net::CascadeCopy is designed to propagate files as fast as possible. Instead of a hub and spoke arrangement (a server copies the files out to all of its clients), this module turns every client into a server. As the doc says, "Once the file(s) are [*sic*] been copied to a remote server, that server will be promoted to be used as source server for copying to remaining servers. Thus, the rate of transfer increases exponentially rather than linearly." Important: Be careful with this thing. If you melt down your entire network using this module, the Secretary and I will disavow any knowledge of your actions.

Take care, and I'll see you next time.