COLUMNS

Practical Perl Tools

Get Your Health Checked

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

Every once in a while I like to inject a little reality into this column, more specifically my reality. This month, instead of writing about some abstract technology or documenting a done deal, I thought it might be fun to work together on a small project that is actually in flight as I write this. This will give you a chance to listen in on my current thoughts (such as they are), and together we can examine some rough code that implements these ideas.

The project I have in mind revolves around a new LDAP cluster that we are currently installing. LDAP stands for Lightweight Directory Access Protocol and is basically the de facto standard for talking to a directory server. Directory servers are used to provide the backbone for most authentication/authorization setups. For example, if you log into a machine that uses some sort of central authentication scheme, chances are the client is doing an LDAP operation at some point as part of the process. This is truly cross-platform (e.g., if you log into a Windows network, you'll be talking LDAP at some point to your ActiveDirectory server(s)).

If you've never dealt with LDAP before, never fear, we won't be assuming much knowledge of it nor will we go very deep. There's a lot that can be written about it (and, indeed, I have a whole chapter and an appendix on it in my book). For the purpose of this column, I'll try to provide enough context so the code makes sense. And, actually, if you take a step back and squint at this column from a little distance away, you'll find that LDAP is just a small detail in the larger picture of health checks, the true subject for today.

So what's a health check and why do I (and maybe you) care? In my case, the LDAP cluster we have set up consists of four LDAP servers that are "behind" a load balancer (actually a pair of them, but that's another story for another column). The load balancer's job is to transparently take in LDAP requests and parcel them out to the actual servers in a balanced way so the load is spread evenly amongst the operational machines. The key word for this column has just been spoken: "operational." One other key purpose for using a load balancer is to make sure that if a machine in a cluster becomes dysfunctional, the clients of that cluster don't notice because the load balancer has cleverly removed that machine from the list of servers it is sending traffic to. If and when that machine returns to service, the load balancer may decide to bring it back into the fold.

Here comes the rub: A load balancer has to know which machines it stands in front of are working and which are not. The way this is typically done is to have the load balancer continuously perform a "health check" on each of the cluster members. Health checks can be simpleminded and naive or fiendishly clever. Right now our current health checks are barely the former, and that's the problem. At the moment, the load balancing software (keepalived, if you are curious) is just checking to see if it can connect to the LDAP port on each of the servers. That's not good enough—we can do much better. Let's rough out a few ways we can improve the situation.

Do What I Do

Being able to connect to the server is great and all that, but LDAP clients connect for a reason. They expect to be able to talk to the server and perform LDAP operations. When writing health checks for almost any service, you'll be off to a great start if your checks mimic even a minimal set of operations a client would be expected to perform. In the case of LDAP this set includes an LDAP bind operation (think of it as "logging into the server"), an LDAP search operation, and an LDAP unbind (which the RFC describes as "the 'quit' operation…the client, upon transmission of the UnbindRequest, and the server, upon receipt of the UnbindRequest, are to gracefully terminate the LDAP session."). Let's look at a little Perl code that does all three things:

```
use strict;

use Net::LDAP;

my ( $server, $binddn, $bindpw, $lookup ) = @ARGV;

my $ldap = Net::LDAP->new($server) or die "Can't connect: $!";
print "connected.\n";

my $res = $ldap->bind( $binddn, password => $bindpw );
$res->code && die "Can't bind: " . $res->error;
print "bound to server.\n";

$res = $ldap->search(
    base   => 'ou=people,dc=example,dc=edu',
    scope  => 'one',
    filter => $lookup,
);
$res->code && die "Search failed: " . "$res->error";
print "entries found " . $res->count . "\n";

$res = $ldap->unbind;
$res->code && die "Unbind failed: " . "$res->error";
print "unbound to server.\n";
```

To quickly walk you through the code, we create an Net::LDAP object that connects us to the server. We then bind() (login) to it. At this point, we execute a search that starts at a particular place in the tree (base), looks at only the part of the tree one level down under that place (one), and filters the result. Lastly, we unbind() to the server. Here's what happens when we run the code:

```
$ ldap.pl localhost 'managerdn' 'managerpw' '(sn=smith)'
connected.
bound to server.
entries found 11
unbound to server.
```

Here you can see we're testing just a few LDAP operations. There are definitely others (compare and modify come to mind) that we should add to this test. More on that last one later. I should also note that this is very simple code that doesn't take into account slow or hung servers (ideally, we should build timeouts into the script to cause it to abort if operations take too long).

If we wanted to be a little cooler, we could go to the logs of a running version of the service and pull a representative slice of the live workload and use it to form the basis of an even better test. Note I said "basis," because we probably don't want to replay it verbatim to our servers, especially if it contains write operations. It would be more than a little embarrassing to have our health checks repeatedly overwrite live data in our directory, though it wouldn't surprise me if this has happened before.

Ah, But How Fast Did I Do It?

Once we know how to pretend to be a client of the server and perform the same operations it might perform, a logical step forward is to model another thing we can expect from our clients: impatience.

In the last section we concerned ourselves with whether our service would answer the phone, reply to our request, and then hang up properly. LDAP clients care about all of these things, but they also care about how long those things take. In many cases a server that replies too slowly might as well be down ("you are dead to me"). Our health check needs to catch this case as well. The first step towards this is timing how long each operation takes. We can do that with code that looks a bit like this:

```
use Net::LDAP;
use Time::HiRes qw(time);

my $start = time();
my $ldap = Net::LDAP->new($server) or die "Can't connect: $!";
my $end   = time();
print "connected: " . ( $end - $start ) . "\n";

$start = time();
my $res = $ldap->bind( $binddn, password => $bindpw );
$end = time();
$res->code && die "Can't bind: " . $res->error;
print "bound to server: " . ( $end - $start ) . "\n";
```

In the above sample we are using the module Time::HiRes because the Perl's native time resolution is seconds (i.e., time() returns the number of seconds since the epoch). In this breakneck world we live in, we expect response times in less than a second. Time::HiRes gives us the extra resolution we need. Take a look at the difference between what time() returns without and with Time::HiRes loaded:

```
$ perl -e 'print time(),"\n"'
1406570529
$ perl -e 'use Time::HiRes qw(time);print time(),"\n"'
1406570567.63434
```

Once we know how fast an operation is, we can then enforce a standard in our health check. Something easy like:

```
($bind_time < 1.0) ? 'up' : 'down';
```

Now, on my unloaded server, that standard is far too generous. Here's what the new code that prints how long each operation takes shows when I run it against an unloaded server:

```
connected: 0.00156688690185547
bound to server: 0.00202512741088867
entries found 11: 0.0248799324035645
unbound to server: 0.000317096710205078
```

Oversimplification Alert! Taking a server out of service based on just a single slow operation sounds both draconian and inefficient. It's more likely you would be better served if you did some math to determine whether the server is consistently reporting back times within a reasonable range. This requires some sort of persistent state be kept around between health checks, a topic we're not going to touch on in this column.

Yup, Still Me in the Mirror

The previous mention of the LDAP modify operation (and write operations in general) brings up another useful aspect to consider when writing health checks. One important way to test a service that includes write operations is through "round trip" tests. Sure, we could write code that performs an LDAP modify of the data and then believe the server if it reports back a successful modification, but it would be far better if we actually did another read to confirm it worked. As the Russian proverb says, "trust, but verify." The idea of round-trip verification comes in handy in many places. For example, when health-checking a mail system, it would be great to have the health check send mail to the system and then attempt to retrieve it a few moments later.

In our case, we can do something like this:

```
my $testdn = 'uid=canaryuser,ou=people,dc=example,dc=edu';
# ...connect and bind as usual, then
$start = time();
my $res = $ldap->modify( $testdn,
                 replace => { 'displayName' => $start } );
$res->code && die "Can't modify: " . $res->error;

$res = $ldap->search(
    base   => $testdn,
    scope  => 'base',
    filter => "(displayName=$start)",
);
$end = time();

$res->code && die "Search failed: " . $res->error;
print "entries found " . $res->count . ": " .
                              ( $end - $start ) . "\n";
```

In this code we modify the value of the displayName attribute in a test user's LDAP entry—we set it to be a timestamp. The next code section attempts to search for that user with a filter that should only return back an entry if the displayName is set to that timestamp correctly. If we return an entry, success. If not, sad trombone.

By the way, a more efficient way to check whether the displayName value has been set to the desired timestamp would be to use a compare operation instead of a search:

```
use Net::LDAP::Constant qw(LDAP_COMPARE_TRUE
                           LDAP_COMPARE_FALSE);
$res = $ldap->compare( $testdn,
                       attr => 'displayName',
                       value => $start);
print "compare succeeded"
       if ($res->code == LDAP_COMPARE_TRUE);
```

In this section we've seen one very simple round-trip test. I'll mention a slightly more sophisticated one related to this test at the end of this column.

Tell Me How You Feel

A piece of well-instrumented server software has a way of reporting its internal sense of health. In the case of the LDAP server we are using (OpenLDAP), it provides a special LDAP suffix we can query to return all kinds of internal counters and statistics. Here's some code that dumps one interesting set:

```
use Net::LDAP;

my ( $server, $binddn, $bindpw ) = @ARGV;

my $monitordn = 'cn=Operations,cn=Monitor';

my $ldap = Net::LDAP->new($server) or die "Can't connect: $!";

my $res = $ldap->bind( $binddn, password => $bindpw );

$res->code && die "Can't bind: " . $res->error;

$res = $ldap->search(
    base   => $monitordn,
    scope  => 'one',
    filter => '(objectClass=*)',
    attrs  => [ 'monitorOpInitiated', 'monitorOpCompleted' ],
);
$res->code && die "Search failed: " . $res->error;

my @operations = $res->entries;
foreach my $operation (@operations) {
    my $dn = $operation->dn;
    my ($opname) = $dn =~ /cn=(\w+),/;
    print "$opname: "
       . $operation->get_value('monitorOpInitiated')
       . " initiated, "
```

```
        . $operation->get_value('monitorOpCompleted')
        . " completed\n";
}

$res = $ldap->unbind;
$res->code && die "Unbind failed: " . $res->error;
```

When run on a pretty fresh server, we get results that look like this:

```
Bind: 34 initiated, 34 completed
Unbind: 25 initiated, 25 completed
Search: 29 initiated, 28 completed
Compare: 3 initiated, 3 completed
Modify: 3 initiated, 3 completed
Modrdn: 0 initiated, 0 completed
Add: 0 initiated, 0 completed
Delete: 0 initiated, 0 completed
Abandon: 0 initiated, 0 completed
Extended: 0 initiated, 0 completed
```

Once you can figure out just which statistics are important to you, it is easy to write a health check that uses them as an indicator of health (with or without the kind of math we discussed in the timing section above). Perhaps you consider a server healthy if it has a small ratio of Modify to Search operations; too many writes could indicate a problem. A query like the one above can determine whether this condition is being met. One last note before we move on: If your server isn't well-instrumented, get a better server (if you can).

Happy Family

For the last section, let's pull the camera back a little further. In a multiple server setup where the servers keep themselves in sync with each other like we have, replication status (i.e., is the data in all of the servers in sync) can be pretty important. So important, we should have a health check for that. OpenLDAP provides a fairly simple mechanism for this. Each time a replication takes place, the server sets an operational attribute called contextCSN with data about the most recent entry that this server contains (it can also keep track of the latest entries it has seen from its replication partners). We can compare contextCSN in two servers to determine whether they are in sync. The structure of this attribute (as per the docs) is:

```
GT '#' COUNT '#' SID '#' MOD

GT: Generalized Time with microseconds resolution,
without timezone/daylight saving:

YYYYmmddHHMMSS.uuuuuuZ

YYYY: 4-digit year (0001-9999)
mm: 2-digit month (01-12)
dd: 2-digit day (01-31)
HH: 2-digit hours (00-23)
MM: 2-digit minutes (00-59)
SS: 2-digit seconds (00-59; 00-60 for leap?)
.: literal dot ('.')
uuuuuu: 6-digit microseconds (000000-999999)
Z: literal capital zee ('Z')

COUNT: 6-hex change counter (000000-ffffff); used to
distinguish multiple changes occurring within the same time
quantum.

SID: 3-hex Server ID (000-fff)

MOD: 6-hex (000000-ffffff); used for ordering the modifications
within an LDAP Modify operation (right now, in OpenLDAP it's
always 000000)
```

Here's a sample set of them from one of my servers:

```
$ ldapsearch -x -LLL -H ldap://localhost
    -s base -b 'dc=example,dc=edu' 'contextCSN'
dn: dc=example,dc=edu
contextCSN: 20140729211439.000593Z#000000#001#000000
contextCSN: 20140514223302.072724Z#000000#002#000000
contextCSN: 20140514224132.047675Z#000000#003#000000
contextCSN: 20140514231128.299773Z#000000#005#000000
```

We could parse this in Perl either with a regular expression like the following (which I found in the Nagios plugin at ltb-project.org):

```
m/(\d{14})\.?(\d{6})?Z#(\w{6})#(\w{2,3})#(\w{6})/g;
```

or by using unpack(), as in:

```
unpack("A14 A1 A6 A1 A1 A6 A1 A3 A1 A6");
```

Parsing these values gives us the time of the latest entry on this server and the latest entry this server has seen from the other servers. If we then go query the other servers, we can start to compare the contextCSN values and get a sense of how in sync they are. On a busy cluster with lots of write activity, you would expect the numbers to drift apart some.

For health check purposes, the question then becomes: How big a difference between servers is acceptable to you before you declare a server "not in sync"? Calculating the difference between times is just a matter of subtraction (perhaps wrapped in an abs() to get the absolute number). As we did before with the timing question, we can then compare it against an acceptable range (or at least an acceptable upper bound).

The key thing here is we are now determining health of a server by its relationship to other servers, a pretty big leap in our thinking. That leap might lead you to revisit the round trip idea from an earlier section. It's not hard to envision a round-trip test where you attempt to see how quickly a write made to one server appears on another (or several, or perhaps all?) replicated server(s). Hopefully, this idea shows you there are a ton of directions we could continue to explore around the simple idea of a health check.

Take care and I'll see you next time.

Endnote: Lest you think this isn't a true reflection of my reality, while working on the section about inter-server synchronization, I realized much to my chagrin that the servers in the LDAP cluster I was building were not properly keeping themselves in sync (they were constantly doing a full synchronization, which is not the way they are supposed to work). Two days of blood, sweat, and tears later, I now have a much better understanding of the role contextCSN plays in replication and how it is supposed to work. (Oh, and the cluster is fixed, too.) Thanks *;login:* column!