Clusterpunch: Distributed Cluster Resource Monitoring

Martin Krzywinski

Monitoring clusters requires specialized tools that can collect, analyze, and present practical statistics in an interactive or scripted manner (Big Brother, Big Sister, Ganglia). Tools like these can help administrators maintain control and awareness while providing users with maximum flexibility, educating everyone about the many ways to get their jobs finished faster, and assisting those who are not using load-balancing systems to distribute their jobs evenly across the network.

Article
Listing 1

jul2003.tar

Clusterpunch, which can be found at:

```
http://mkweb.bcgsc.ca/clusterpunch
```

is a distributed execution system developed to provide administrators and users with a means to measure the computational availability of cluster nodes. Distributed mini-benchmarks (punches) are used to measure computational availability and rank cluster nodes according to how quickly they can compute a short task, like write a file or crunch some numbers. Various types of benchmarks can be used to rank the nodes, thereby providing a way to rate specific subsystems (CPU, I/O, memory, network). For example, if a user were most interested in performing heavy I/O operations, a customized punch would be used to find the nodes whose I/O is fastest (these nodes may not necessarily house the fastest CPUs or be the least busy at the time).

Implementation Overview

Clusterpunch is written in Perl and uses UDP to send punches and receive results from participating nodes. It is designed to run with minimum requirements -- only Perl and a few common modules are needed. A daemon, clusterpunchserver, runs on every node to be monitored and listens to a predefined UDP port (default 8095). Client utilities use the API defined in the Perl module clusterpunch.pm. The API provides the means to use UDP broadcasts to send punches to all listening clusterpunchservers on the network and to retrieve the results. The punches are sent using a formatted string command, "punch1;punch2(arg);...", which includes the names of the punches along with any arguments. The response is a serialized hash, in which the keys are the punch statistic names and the values are the punch results, "punch1=>1.5;punch2=>0.5;...".

Punches are defined by parameters, which govern the way a punch is evaluated, and by Perl code or system commands, which defines the task for the punch. A punch may be associated with a system call, thereby allowing the use of third-party benchmarking or diagnostic binaries. All punches are defined in an Apache-like configuration file. Support for multiple configuration files allows nodes to carry out different tasks for a given punch call.

In the Clusterpunch distribution, a global configuration file defines the punches and every node runs this same punch code. Benchmark punches for testing CPU, memory, and I/O subsystems are defined, along with a few diagnostic punches. If your hardware profile is similar for all nodes then the default settings will likely provide accurate relative ranking. However, if you have a heterogeneous environment, particularly with different CPU types that may have their own particular strengths (e.g., float vs. integer arithmetic), you can easily define your own punches and use a host-specific or host group-specific configuration. The benchmarking implemented in Clusterpunch is designed to rank nodes, not to measure absolute performance rating. You can use Clusterpunch to track the performance rating of your entire cluster, but you cannot compare this value to anything other than another Clusterpunch performance rating obtained with the same punches. If you are looking for an absolute measure of performance, look to HPL, the High-Performance Linpack benchmark used in the Top 500 list rankings.

The following utilities are available in the current distribution:

- clusterlogin -- rlogin into the highest ranked node.
- clustersnapshot -- Show rankings for all nodes.
- clusterbench -- Compute the overall availability for all nodes.
- clusternodecount -- Count all listening nodes.

Implementation Details

clusterpunchserver

The daemon utility clusterpunchserver (Listing 1) runs on every node. The server and all client utilities first load one or more Apache-like configuration files, which are parsed by the excellent Config::General module written by

Thomas Linden. The configuration file syntax is explained below. Once the configuration file(s) are parsed, the clusterpunchserver opens a UDP socket on the specified port and begins to run in the background, the preferred mode in a production environment. In the main loop, the server listens on the socket for any incoming messages. Once a message is received, it is parsed into a list of hashes, with each list element being {command=>COMMAND, args=>[arg,arg,...]}. If any of the commands is "reload", the server reloads all configuration file definitions before forking off a child to process the command list. This allows you to alter your configuration on the fly without restarting the servers. If any of the commands is "shutdown" or "halt", the server shuts down.

A child process is forked to handle the incoming punch requests. For each command name, a punch of the same name is executed if found in the configuration file. A command has no effect if there is no corresponding punch:

```
my @punches = @{$CONFIG{punch}};
PUNCH: foreach my $command (@commands) {
    my $commandtext = $command->{command};
    my ($punch) = grep($_->{name} eq $commandtext, @punches);
    next PUNCH unless $punch;
    ...
}
```

If the punch is found, the child performs the task defined in the punch. There are two possibilities here:

```
if($function) {
     @_ = @args;
      $call_value = eval $function; # evaluate Perl code
} elsif ($system) {
     open(PROC,"$system |"); # make a system call
     while(<PROC>) {
     s/^\s+//;
     $call_value .= $_;
     }
     close(PROC);
     chomp $call_value;
}
```

If the punch has a function parameter defined, which is expected to hold Perl code, then the code is executed by "eval". If the punch has a system parameter defined, which is expected to hold a system command, a pipe is opened to that process and all output is captured. For convenience, all leading spaces and the last newline are stripped out.

The punch returns either the return value of the punch code (if "valuetype = return" in the configuration file) or the time taken in seconds to run the code (if "valuetype = timer" is used). The timing is done by the Time::HiRes module by Jarkko Hietaniemi. Typically, benchmark punches will be timed and diagnostic or resource discovery punches will return values (e.g., amount of free memory):

```
if($valuetype eq "return") {
  $punch_value = $call_value;
} else {
  $punch_value = tv_interval($timer); # Time::HiRes
}
```

It is possible to re-map the punch value using the function defined in "valuemap", thereby filtering the output of the punch without altering the punch code. This is particularly useful for punches that call external binaries:

```
if($valuemap) {
  @_ = ($punch_value);
  $punch_value = eval $valuemap;
}
```

The return value of the punch, possibly filtered by "valuemap", is added to the %STAT hash, which will be eventually returned to the client. The punch's "statistic" field defines the name of the key that will hold the value. A punch may also define a "cumulative" field to which the value will be added. The cumulative statistic is designed to conveniently store sums of values of other punches:

```
$STAT{$statistic} = $punch_value;
$STAT{$cumulative} += $punch_value if defined $cumulative;
```

Finally, the %STAT hash is populated with the default 'live'=>1 entry, as well as with the "host" key that stores the name of this node. If the command passed to the node is an empty string, then the "live" and "host" values will be the only populated hash entries. The "live" key is meant to facilitate counting the nodes by summing the punch return values, rather than keeping your own count index. The last act of the child, before it exits, is to send the serialized %STAT hash back to the server:

```
$STAT{live} = 1;
$STAT{host} = $SERVERNAME;
my $dump = Dumper(\%STAT);
$sock->send(pack("a*",$dump));
exit;
```

Throughout the clusterpunchserver code, the logging function Log() is called to write informational text to the node's log file. Except for debugging or development purposes, logging should be disabled because it can generate a lot of repetitive content.

clusterpunch.pm

All client utilities that communicate with the clusterpunchserver use the API defined in clusterpunch.pm. The API module is responsible for providing functionality for finding and parsing the configuration files, sending and receiving messages, and processing received messages.

Configuration files are loaded in LoadConfig(). Once the file list has been determined, each file is parsed with the Config::General module and parameters are stored in a %CONFIG hash. The %CONFIG hash may have some of its previously defined values replaced, making it possible to use multiple files to differentiate between network-wide and host-specific parameters:

```
foreach my $configfile (@configfiles) {
    my $conf = new Config::General(-ConfigFile=>$configfile,
                                   -LowerCaseNames=>1,
                                   -AutoTrue=>1,
                                   -UseApacheInclude=>1);
    %CONFIG = (%CONFIG,$conf->getall);
}
```

The communication with each clusterpunchserver is mediated by the ClusterPunch() function. Here, a UDP socket is opened and set to broadcast mode. The text command (e.g., "punch1;punch2(10)") is sent over the socket:

```
my $sock = IO::Socket::INET->new(Proto=>"udp",PeerPort=>$port)
$sock->sockopt(SO_BROADCAST() => 1) if $host =~ /255$/;
my $dest = sockaddr_in($port,inet_aton($host));
send($sock,$command,0,$dest);
```

Over the next $timeout seconds, the client collects all incoming responses, unpacks the message, evaluates encoded data structure, and adds the result to the %RESPONSE hash, keyed by the responding node name:

```
my %RESPONSE;
eval {
    local $SIG{ALRM} = sub { die "timeout\n"; };
    alarm($timeout);
    while(1) {
      next unless my $addr = recv($sock,$data,MAX_MSG_LEN,0);
      chomp($data);
      my ($port,$peer) = sockaddr_in($addr);
      my $host = gethostbyaddr($peer,AF_INET) || inet_ntoa($peer);
      $host =~ s/(.*?)\..*/$1/g;
      my $datadump = unpack("a*",$data);
      my $STAT1; # STAT1 is defined in the structured output of Data::Dumper
      eval $datadump;
      $RESPONSE{$host} = $STAT1;
    }
  };
  if($@) {
    die $@ unless $@ eq "timeout\n";
  }
  return %RESPONSE;
```

The %RESPONSE hash has the structure:

```
{
 nodename1=>{punch1=>value,...,live=>1,host=>nodename1},
 nodename2=>{punch1=>value,...,live=>1,host=>nodename2}.
 ...
}
```

This hash contains all the necessary information to derive node ranking, with the help of punch parameters defined in the configuration file(s). The punch parameters need to be read in by the client because, a priori, it is not known how

a node's rank is to be computed from the result of a particular punch. For example, if the punch result is the MHz rating, higher values are better, but if the punch result is a timed benchmark, lower values are better.

Processing of the %RESPONSE hash is done in the ProcessResponse() function. This function can return a formatted, sorted table of results to STDOUT or return the host names of the top N nodes ranked by a punch result.

clusterpunch.conf

The punches are defined in this configuration file, parsed by Config::General, where parameters and parameter blocks are defined:

```
parameter = value

<blockname>
parameter1 = value
parameter2 = value
code <<END
  ...Perl code...
END
</blockname>
```

In addition to a custom configuration file passed via command-line parameter, multiple configuration files can be read from:

```
~/.clusterpunch
 ../etc/clusterpunch.conf (relative to location of binary)
 /usr/local/etc/clusterpunch.conf
 /etc/clusterpunch.conf
```

The files are parsed in the order shown above, so that the configuration in the host-specific /etc/clusterpunch.conf overrides any definitions in global files. The configuration file contains three sections: definition of parameters, definition of sort blocks, and definition of punches. The supported parameters are:

- logdir -- Directory to which to write node logs
- logging -- Toggle logging
- verbose -- Toggle verbose STDOUT messages from server
- daemon -- Run in background node
- debug -- Extra debugging
- port -- UDP port to use
- broadcast -- Address to use by client utilities
- timeout -- Amount of time to wait for responses from nodes

Sort blocks define the way statistics, such as cumulative statistics, are displayed and how they are to be sorted in the determination of node ranking. The "b_all" statistic stores the sum of the benchmark times from the CPU, I/O, and memory punches. Because it is a compound statistic, not associated with a single punch, its sort and format are defined separately in a sort block.

```
<sort>
      statistic = b_all
      sort = ascending
      format = %6.3f
</sort>
```

Punches are defined in a similar fashion. Consider, for example, a sample punch in which a node makes 1 million calls to rand(). The name of the punch is punch1, as is the statistic the punch uses. The name of the punch and its statistic do not need to be the same. The punch return value is the time it took to execute the code (set by "valuetype"). As an example, a value filter is defined by "valuemap" and the returned value is the square root of the time:

```
<punch>
name = punch1
statistic = punch1
valuetype = timer
valuemap = sqrt(abs($_[0]))
format = %6.3f
function <<CODE
for (my $i=0;$i<1e6;$i++) { rand () }
CODE
</punch>
```

A punch may also be associated with a system call, such as below:

```
<punch>
name = punch2
statistic  = dirlist
valuetype  = timer
system = "/bin/ls -alR /etc &> /dev/null"
</punch>
```

Installation

The latest distribution is available at:

```
http://mkweb.bcgsc.ca/clusterpunch
```

To install and run Clusterpunch, you will need Perl and the Config::General module. It's likely that you already have Data::Dumper, FindBin, Getopt::Std, IO::Socket, and Time::HiRes, which are also required:

```
> tar xvfz clusterpunch-x.xx.tgz
> cd clusterpunch-x.xx
> less README
```

Running the benchdriver utility runs all the punches at the command line. This utility is useful when debugging and customizing the default behavior of the punches to suit your environment. The output of benchdriver shows that node 0of8 took 0.55s to perform the CPU benchmark:

```
> bin/benchdriver
punch1       0of8 0.420704
punch2       0of8 0.077519
benchmem     0of8 0.786146
benchio      0of8 2.014951
benchcpu     0of8 0.551969
mhz          0of8 2792
load         0of8 0.08
uptime       0of8 92.1664930555556
nusers       0of8 6
jobusers     0of8 mapper:0.00 martink:0.18 phuang:0.00
lsof         0of8 666
date         0of8 13:58:03
nrunning     0of8 1
```

Starting the daemons is done by using rsh, iterating over all hosts in hosts.dat. Keep in mind that any benchmarks will be done at the nice value of the daemon:

```
> clusterpunch.start
```

You can now use the client utilities to communicate with the nodes. Counting the nodes is done with clusternodecount:

```
> bin/clusternodecount
59
> bin/clusternodecount -v
0of0
1of0
...
7of8
8of8
```

A formatted table of punch results can be obtained with clustersnapshot. In the example below, three subsystem punches and the mhz punch, which returns the MHz rating, are used. The client will wait 20 seconds for responses and the results will be sorted by the "b_all" statistic. The benchio punch is passed two arguments specifying that two 60-Mb files should be written to a local disk (/tmp) for this I/O benchmark:

```
> bin/clustersnapshot -c "benchmem;benchio(60000,2);benchcpu;mhz;load" -t 20 -s "b_all"

        host  b_all  b_cpu   b_io  b_mem live load   mhz
        4of2  2.215  0.722  0.588  0.905    1  2.1  1992
        8of1  2.215  0.722  0.590  0.904    1  2.0  1992
        9of0  2.217  0.724  0.587  0.906    1  2.0  1992
        9of1  2.217  0.721  0.590  0.906    1  2.0  1992
        4of0  2.222  0.724  0.586  0.911    1  2.1  1992
```

```
      7of0  2.223  0.724  0.592  0.908    1  2.0  1992
      6of0  2.225  0.724  0.595  0.906    1  2.0  1992
      0of1  2.232  0.716  0.600  0.916    1  0.0  1992
      5of0  2.233  0.724  0.600  0.909    1  2.0  1992
     ...
TOTAL 187.519 38.969 95.558 52.991   59 83.1 140938
```

Measuring Cluster Resources

The clusterbench utility returns the total computational availability of all nodes. The output has the two-line format suitable for direct input into MRTG:

```
> bin/clusterbench -t 20
206
140
```

The two values that are returned are the cluster resource rank and the sum GHz of all nodes. Clusterpunch defines available resources as inversely proportional to the benchmark time. If a node's benchmark time is t, then its resource rating is k/t, for some scale factor k (default k=10). Using the k/t measure, a node that completes a benchmark in half the time has twice the resource rank.

The resource rank for a collection of nodes is proportional to the sum of the resource rank of each node. Given nodes n1,n2,...,nN, each with a resource rank of k/t1,k/t2,...,k/tN, the cluster's rank is:

```
R = k/t1 + k/t2 + ... + k/tN
```

The value R will increase if you have more idle nodes, add more nodes, or upgrade nodes to contain faster subsystems. The value will decrease if nodes become sluggish because of other jobs, nodes go offline, or your IT manager asks you to downgrade your CPUs. Because each node returns detailed benchmark times, you can compute the node's ranking for each subsystem, k/tCPU, k/tIO, k/tMEM, and compute cluster resource rank for subsystem S using:

```
RS = k/tS1 + k/tS2 + ... k/tSN
```

Summary

Clusterpunch is a lightweight and portable system for running distributed mini-benchmarks (punches) across a cluster in order to rank the computational availability of the cluster's nodes. Punches are stored in an external configuration file and can be defined using Perl code or associated with a system call and may be customized to simulate the type of load in your environment. A Perl API is provided to incorporate this functionality into your own scripts.

Resources and References

Big Brother -- **http://bb4.com/**

Big Sister -- **http://bigsister.graeff.com/**

Clusterpunch -- **http://mkweb.bcgsc.ca/clusterpunch**

Ganglia -- **http://ganglia.sourceforge.net/**

Config::General -- **http://search.cpan.org/author/TLINDEN/Config-General-2.15/General.pm**

MRTG -- **http://people.ee.ethz.ch/~oetiker/webtools/mrtg/**

RRD -- **http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/**

PBS -- **http://pbs.mrj.com/**

HPL -- **http://www.netlib.org/benchmark/hpl/**

Top 500 -- **http://www.top500.org/**

Martin Krzywinski is a bioinformatics scientist at the Genome Sciences Centre. He spends a lot of time applying Perl to munge through biological data and to create data analysis pipelines. He can be reached at: ***martink@bcgsc.ca****.*