Pong: A Flexible Network Services Monitoring System

Helen E. Harrison, Mike C. Mitchell, and Michael E. Shaddock – SAS Institute, Inc.

ABSTRACT

In distributed computing environments it is important to determine not only whether individual critical machines are up or down, but also whether the individual services they offer are available. Our site was using a network monitoring package which used ICMP ECHO packets to determine when individual network components were unavailable. We found that there were many occasions where a server would reply to the ICMP ECHO, but would still not be providing the services it should be providing. We needed a tool which would let us monitor high level services such as AFS or NFS file service. This paper will describe a tool called `pong` which was developed to meet these needs. `Pong` is a highly configurable monitoring tool which ''pings'' individual services at predefined intervals and executes appropriate actions when the state of that service changes. We use `pong` to monitor three or more services on each of 110 servers.

Introduction

SAS Institute Inc. is a privately owned software development company whose primary software development environment is a network of 1500 Hewlett Packard 700 series workstations and servers [1]. These machines share data using a combination of AFS and NFS file systems over an aggressively subnetted TCP/IP network. During the second half of 1992 we experienced increasing occurrences of failures in services on key servers which were not being detected by our commercial network monitoring package. The monitoring package used only ICMP ECHO packets to determine if a machine was operational. We found that the use of ICMP ECHO packets never gave a false negative, but did give false positives. The HP servers would occasionally fail in a way that they would still repond to pings, but otherwise would appear to be unavailable. In addition we found situations where the AFS or NFS daemons would stop responding to requests but otherwise the machine would be healthy. We needed a monitoring system which would allow us to determine the state of a system and the services it was providing. Another requirement we have for a monitoring system is that it be configurable and easily extensible. For example, when a failure is detected we may want to send mail, page someone, send a zephyr-gram, log the failure, or whatever else we might decide we need. We want to control how often a check is performed, what type of check, and how many replies are ''dropped'' before a service is considered unavailable. `Pong` addresses these requirements. There have been number of other projects which also addressed these problems, including the emerging SNMP based technologies, and many of these show great promise. So far these alternatives have not provided the functionality, efficiency, and cost effectiveness that would make them appropriate replacements for `pong` for service level monitoring.

The `Pong` Program

`Pong` has three basic characteristics: service monitoring routines, monitoring intervals and response actions. `Pong` monitors these services by constructing a UDP packet and sending it to the appropriate port on the remote machine and then watching for a response. The service monitoring routines include ICMP ECHO requests which measures basic network connectivity, the `inetd` ''echo'' service which measures `inetd`'s responsiveness (a good measure of the overall health of a machine), NFS nullRPC messages which measures NFS responsiveness, and AFS RX DEBUG requests which measures AFS's responsiveness. These functions simply return success or failure. Each of these built-in monitoring functions were implemented using existing functionality and required no changes on the systems being monitored. Response actions are executed when a machine changes state. `Pong` does not have any built-in notification routines. Its sole job is to monitor the state of the specified machines leaving notification of state changes to other programs.

If a monitored service becomes unresponsive, `pong` starts checking with increased frequency. After the specified number of ''down'' checks are performed and the service is still unresponsive, `pong` executes the specified ''down'' response action. It will then switch back to checking less often, until the service responds. After the first positive response it starts checking more often until the specified number of ''up'' checks are performed. At that time it executes the specified ''up'' response action, and switches once more to checking less often (see Figure 1).

Configuration

The pong configuration file lists the type of service to monitor (icmp, inetd, nfs, or afs), the frequency with which to check the service, the command to execute when the service is discovered to be unavailable, the command to execute when the service is available again, and finally the list of machines on which to monitor these services (see Example 1).

The first line of the configuration file specifies the type of service, the time between checks when it knows the service status, the time between checks when it isn't sure of the status, the number of consecutive lost responses before a service is marked down, and the number of consecutive received responses before a service is marked up. The next two lines list the command to be executed when the the service is marked down and marked up, respectively. The following lines list all of the machines this type of check will be performed on. It will accept hostnames, IP addresses, and hostclass specifications [2].

The up and down command lines will be passed to /bin/sh, so they can full advantage of UNIX regular expression functionality. Pong replaces ''%h'' on the command line with the host name (or IP address) of the failing service. The ''%t'' and ''%T'' character sequences are replaced with the time the service went down and the current time, respectively. The time is converted to the UNIX ctime format, without the newline on the end (see *ctime* (3) manual page).

The usefulness of pong depends greatly on the scripts used for notification. For simplicity this example shows the down command as an echo command piped to a mailer. For our production system we actually have pong execute a perl script. Our perl script writes an informative timestamped line to a log file, sends mail to the system administrator, sends a color-coded Zephyr notification [3] (red for

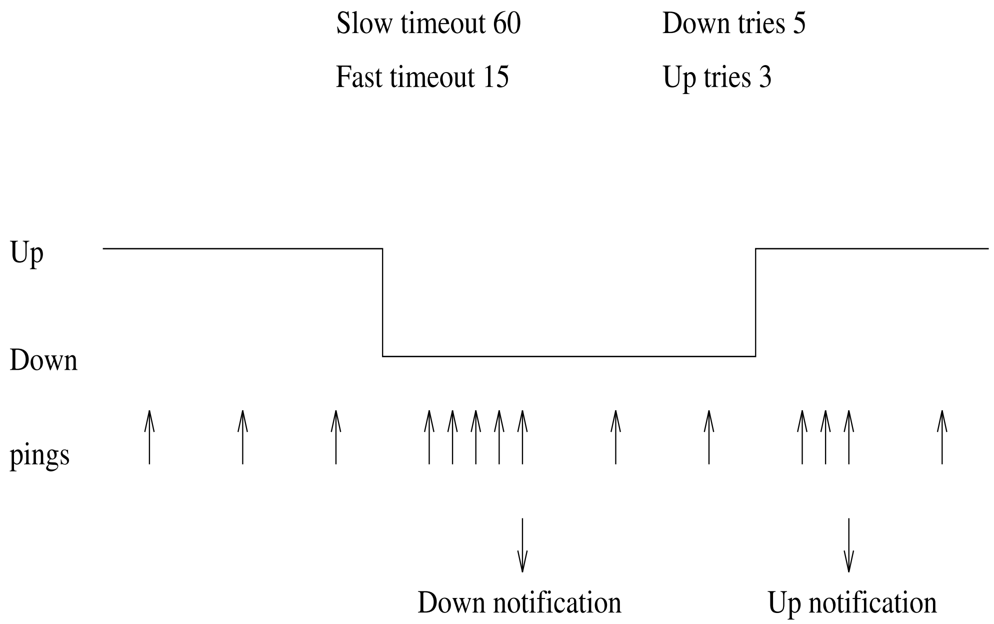

Figure 1: Polling Intervals

```
#
# nfs pings -- send a null rpc to NFS process.
#
#       slow    fast    down    up
#type   timeout timeout tries   tries
nfs     60      5       4       3
echo down "%h: nfs down at %t" | /bin/mail sysadmins
echo up   "%h: nfs up at %T" | /bin/mail sysadmins
dbserv1
152.5.64.10
=code_servers
```

Example 1: A typical entry in the configuration file ''down'' messages and green for ''up'' messages) to an interested party subscription list including our Data Center operators and the systems administrator it has determined to be ''on call''. If that system administrator is not presently logged in, the notification routine will send him mail. We send our messages to the ''servdown'' Zephyr class, using the machine name as an instance. (The Zephyr system manages subscription lists and sends out messages to anyone who subscribes to a specific ''class'' and ''instance'', somewhat similar to newsgroups or mailing lists.) The use of Zephyr allows people who are interested in particular servers to receive notification without requiring intervention by support personnel.

Extending Service Monitors

As described previously, `pong` currently monitors four basic services (NFS, AFS, inetd and ping) and we plan to extend it to monitor other services such as `named` and `zephyrd`. `Pong` is structured so that additional service definitions can be added easily by adding entries to a table of functions. `Pong` constructs a UDP packet based on the function definition and sends it to the specified port. All that is needed is to identify a way to probe a network service so that it sends a packet in response. For example, AFS does not have a predefined ping-type function like the `inetd` echo server. In order to ''ping'' AFS, we construct an invalid AFS RX DEBUG packet to which the AFS server process responds with a packet that says ''this request is invalid''. If there is not a way to probe a service via a UDP packet, it can be added to `pong` by defining a a service under `inetd` that queries the desired service, and have `Pong` poll the new `inetd` service.

When `Pong` starts it initializes an event queue which contains one entry for each machine and ping type combination. The entries also contain a subroutine to call and the time to make the call. After the event queue is initialized, `Pong` loops calling `select` and handling events. The top of the loop checks to see if an event timer has expired, and if so calls the appropriate function. After it handles all the expired timers, it calculates the amount of time until the next timer expires. It passes that time off to the `select` routine as the maximum time to wait for input on a file descriptor. If the `select` routine indicates there are packets ready to be read, the appropriate input routines are called. After all the packets are read the loop returns to the top.

There are only three subroutines that need to be written to add a new ''ping'' type. `Pong` calls an initialization routine at startup, a packet-send routine when a timer expires, and a packet-read routine routine when the `select` call says there is a packet to be read.

The initialization routine is passed a pointer to a ''ping_type'' structure, in which it fills in a file descriptor. Anything else it does is dependent on what type of ''ping'' is being created. Typically the initialization routine opens a socket, looks up what UDP port number to use, and fills in whatever data structure will be used as a packet header. Information that might be needed to send a packet or receive a packet can be stored in global variables.

The packet-send routine is called every time a packet of the appropriate ping-type is to be sent to a host. It is passed a pointer to some host-specific data which includes the IP address of the host, a pointer to the ''ping_type'' structure (which contains the file descriptor), and a unique sequence number for that packet. Typically all the packet-send routine does is put the sequence number into the packet built by the initialization routine, fill in the socket address with the UDP port number and host address, and send the packet.

The packet-receive routine is called after the `select` call indicates there is a packet to be read. It is passed a pointer to the 'ping_type' structure

```
% pongstat nastase
Host          Ping   State         Last Up                   Last Down
------------ -----   ----- ------------------------  ------------------------
nastase       afs    down                            Sun May 22 09:02:47 1994
nastase       inetd   up   Sun May 22 15:25:46 1994  Sun May 22 09:03:01 1994
nastase       nfs     up   Sun May 22 15:25:11 1994  Sun May 22 09:02:26 1994

% pongstat -d
Host          Ping   State         Last Up                   Last Down
------------ -----   ----- ------------------------  ------------------------
seles         nfs    down                            Sun May 22 10:54:44 1994
nastase       afs    down                            Sun May 22 09:02:47 1994

% pongstat -help
usage: pongstat [-p server] [-t type] [-d] [ machine ... ]
```

Example 2: Sample output of the `pongstat` command (which has the file descriptor). Typically a packet is read from that file descriptor and the return address is used to look up the host-specific data. Once the host-specific data has been located, the packet-receive routine can check the received packet to be sure it has the right sequence number, the right format, and so on. If the reply packet is acceptable, the packet-receive routine sets a flag in the host-specific data structure indicating that it received a reply from the packet sent.

A ping-type that uses a virtual circuit instead of datagrams (TCP instead of UDP) would be possible, even though `pong` was designed for datagrams. For a TCP based ''ping'', both the initialization routine and the packet-receive routine would be empty stubs. The packet-send routine would open the socket, make the connection, exchange whatever information is necessary, and close the connection. If the packet exchange indicated the service was functioning, the packet-send routine would set the special ''I received a reply'' flag in the host-specific data structure. The problem with this approach is that it can take a long time for the TCP connection to be established and a long time for the packet exchange. The person writing the code must put in code to abort the connection if it takes more than a few seconds for the the reply to come back, otherwise all of the other pings will be delayed.

Related Utilities

Pongstat

Once we had `pong` up and running, we found that we wanted to be able to determine current state of all the servers and services it was checking. `Pong` writes messages to a log file, but it was not practical to parse through all the messages to find the current status. `Pong` knows the state, so we modified it to dump out the current state when it received a TCP connection to the `pongstat` port. We could then use a command like `telnet` to get a current status report, but we decided it would be simpler to write another program to get the information; we call it `pongstat`.

`Pongstat` uses TCP to connect to the `pong` program. `Pong` sends it an alphabetized list of machines, services, an up/down flag, and the time of the last down-to-up and up-to-down transitions. `Pongstat` reformats the output and prints out only the services and/or machines specified. By using the ''-d'' flag, only the services that are currently down are shown. The sample output in Example 2 shows that a server named nastase is up except for its AFS processes. Everything on that system went down around 9:00 AM and came back up 15:25. The only things currently down are NFS on seles, and AFS on nastase.

King_Pong

The `pongstat` command served our system administrators well, but the our operators wanted a simple way to tell the state of the network at a glance. From that need came `king_pong`, a GUI program that graphically displays the current state of all the servers. It is written using the Tcl, Tk, and Extended Tcl packages. It connects to `pong` once a minute via the `pongstat` port and draws red, green and yellow boxes showing the server status. A green box means that a server is up; a red box means that a service on that servers is down; a yellow box means that a service has gone down and back up again. A yellow state must be acknowledged before returning to green. This ensures that failure conditions which correct themselves will not go unnoticed.

A mouse click on a box pulls up a window that shows the current state, as if `pongstat` was executed for that particular machine. In addition to getting server status with a mouse click, it also has the ability to display additional information about each server, such as which groups use it, network services which it provides, its physical location and special contacts for the server. This information allows us to give general documentation to operations on how to respond to alarms while keeping the more dynamic specific information on each server in a more easily updated online form. `King_pong` also allows the operators to ignore a server, so that if a server needs to be out of production for a couple of days, they will not be seeing alarms while it is out of service.

Related Work

`Pong` has similarities to the `buzzerd` system [4] in implementation and solves similar problems. Through notification programs, `pong` can be configured for notification schemes of similarly complex to `buzzerd`. In our environment, we have a Data Center which is staffed 24 hours/day so that notification programs could be fairly simple. Instead our efforts were concentrated on flexibility of its monitoring capabilities. One significant difference is that `buzzerd` requires that a special daemon be run on a each server that it monitors. `Pong` does not require that anything be changed on a monitored machine which can be a distinct advantage if the target is a PC, router, terminal server, network printing device, or other network device which does not happen to run UNIX. With over 100 servers to monitor, we do not want the overhead of managing an extra monitoring daemon if we can avoid it.

For situations which would require running a special monitoring daemons on each machine, `pong` could still be used as the notification agent. The monitoring daemon could check many different system resources like the `sysmod` component of `buzzerd`, or even be more sophisticated like the `SPA`

Expert System[5]. As long as the monitoring daemon can respond to pong's queries, pong can be used as part of the notification process. Using one notification agent simplifies administration of the many different monitoring tools.

If a network router or bridge fails, pong may try to send 100's of ''down'' messages, even though only a single component is really unavailable. The ''Big Brother'' system[6] addresses this type of problem by introducing dependency lists in its configuration file. With pong this problem can be addressed by using ''smart'' notification scripts. There are some interesting trade-offs here, which for pong were decided in favor of simplicity.

Many of the commercial network monitoring systems are using SNMP as their querying mechanism. Pong does not use SNMP for several reasons: SNMP is not universally available, SNMP is a different protocol with a learning curve of unknown steepness, and SNMP would have required another daemon on each monitored machine. Instead, it was decided to use well known and well understood protocols. Given the extensible nature of pong, we believe it will be relatively simple to add SNMP querying functions.

Conclusions

Commercial network monitoring systems have been emerging over the last few years which show promise of becoming highly sophisticated and useful. Unfortunately they have not reached their full potential and have been expensive for the core functionality that they have offered. Specifically, we have found no commercial tools which will monitor AFS services. Pong was written as a survival tool to fill this vacuum and help us monitor our rapidly growing network. Its simplicity and flexibility have made it a powerful tool.

Availability

For information on the availability of Pong, please send mail to heh@unx.sas.com.

Author Information

Helen E. Harrison is the UNIX Support Manager at SAS Institute Inc., where her group provides hardware and software support for a network of over 1400 UNIX workstations and servers. She has been involved in UNIX systems administration for over 10 years and holds a B.S. in Computer Science from Duke University. Reach Helen at SAS Institute Inc, SAS Campus Drive, Cary, NC 27513; or by e-mail at heh@unx.sas.com.

Mike Mitchell is a Systems Programmer in the UNIX Support Group at SAS Institute Inc. He has been involved in Distributed Computing for over 5 years and UNIX systems for 15 years. He holds a B.S. in Computer Science and a B.S. in Electrical Engineering, both from North Carolina State University. Reach Mike at SAS Institute Inc, SAS Campus Drive, Cary, NC 27513; or by e-mail at mcm@unx.sas.com.

Michael Shaddock is a Systems Programmer in the UNIX Support Group at SAS Institute Inc. He has been involved in UNIX systems administration for over 8 years and holds a M.S. in Computer Science from the University of North Carolina at Chapel Hill. Reach Michael at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513; or by e-mail at shaddock@unx.sas.com.

References

1. Helen E. Harrison, ''So Many Workstations, So Little Time,'' *LISA VI Proceedings*, pp. 79–86, Long Beach, CA, October, 1992..
2. Helen E. Harrison, Stephen P. Schaefer, and Terry S. Yoo, ''Rtools: Tools for Software Management in a Distributed Computing Environment,'' *Proceedings of the Summer USENIX Conference*, pp. 85–93, San Francisco, CA, June, 1988..
3. C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld, ''The Zephyr Notification Service,'' *Proceedings of the USENIX Winter Conference*, pp. 213–219, Dallas, TX, February, 1988..
4. Darren R. Hardy and Herb M. Morreale,, ''buzzerd: Automated Systems Monitoring with Notification in a Network Environment,'' *LISA VI Proceedings,*, pp. 203–210, Long Beach, CA, October, 1992..
5. Peter Hoogenboom and Jay Lepreau, ''Computer System Performance Problem Detection Using Time Series Models,'' *Proceedings of the Summer USENIX Conference*, pp. 15–32, Cincinnati, Ohio, June, 1993..
6. Don Peacock and Mark Giuffrida, ''Big Brother: A Network Services Expert,'' *Proceedings of the Summer USENIX Conference*, pp. 393–398, San Francisco, CA, June, 1988..

Appendix A: Sample Notification Script

```
#!/local/bin/perl

$host = $ARGV[1];
$ptype = $ARGV[2];
$dtime = $ARGV[3];
$ctime = $ARGV[4];
$log = "/local/etc/servdown.log";

if ( $ARGV[0] eq "up" )
{
    $color = "green";
    $msg1 = $host . ":  " . $ptype . " up at " . $ctime;
    $msg2 = "down at " . $dtime;
}
else
{
    $color = "red";
    $msg1 = $host . ":  " . $ptype . " down at " . $dtime;
    $msg2 = "x";
}

open(LOG, ">>$log");
print(LOG "$msg1\n");
close(LOG);

if ( $msg2 ne "x" )
{
    open(MSG, "|/local/bin/zwrite -n -c servdown -i $host");
    print(MSG "@beep()@color($color)@large(@b($msg1))\n");
    print(MSG "@beep()@b($msg2)\n");
    close(MSG);
}
else
{
    open(MSG, "|/local/bin/zwrite -n -c servdown -i $host");
    print(MSG "@beep()@color($color)@large(@b($msg1))\n");
    close(MSG);
}

$onc = "no_one";
open(ONC, "/local/etc/pong.oncall");
while(<ONC>)
{
    chop;
    if (length($_) > 0)
    {
      $onc = $_;
      close(ONC);
    }
}
close(ONC);

$rc = 1;
if ($onc ne "no_one")
{
    $rc = system("/local/bin/zlocate $onc > /dev/null 2>&1");

    if ($rc == 0)
    {
      open(MSG, "|/local/bin/zwrite -n $onc");
      print(MSG "@beep()@color($color)@large(@b($msg1))\n");
      if ( $msg2 ne "x" )
```

```
        {
            print(MSG "@beep()@b($msg2)\n");
        }
        close(MSG);
        $rc = $?;
      }
      if ($rc != 0)
      {
        open(MSG, "|/usr/bin/mailx -s \"server problems\" $onc");
        print(MSG "$msg1\n");
        if ( $msg2 ne "x" )
        {
            print(MSG "$msg2\n");
        }
        close(MSG);
      }
  }
```