Control plane benchmark technical documentation

Heikki Lindholm, Taneli Vähäkangas

Helsinki May 4, 2007
Final
University of Helsinki
Department of Computer science

Contents

1 Introduction

Following our earlier crafted control plane benchmark specification [1], we created an implementation of the benchmark based on readily available components. This document has the details of the implementation, including its usage instructions.

The rest of this document is divided to the following chapters. Chapter 2 contains the implementation notes. Chapter 3 contains the user documentation.

2 Implementation

This section documents the implementation of the benchmark.

2.1 Overview

The benchmark implementation uses two major software packages: OpenSER[1] and SIPp[2]. The implementation has scripts for building, configuring, and running the software from the packages suitably to form a benchmark. The implementation also includes tools for generating and monitoring server load, registering users to OpenSER, analysing the benchmark results, and producing graphs from the results.

OpenSER is used as the telephony server and SIPp as the user-agent clients (callers) and user-agent servers (recipients.) OpenSER is instrumented to output time stamped events to system log from where the benchmark results are derived from. A load generator and load monitor are also run alongside the telephony server on the SUT.

Table 1 lists the files comprising the benchmark package and their roles. The scripts are implemented in bash, Python, and GNU Octave. Log analysis and load generation are implemented in C.

2.2 Build tools

Building the benchmark is automated by two `bash` scripts: `build.sh` and `build-sut.sh`. The scripts are used to build everything required for the benchmark. Building the benchmark could be done by hand as well, but the scripts make building the benchmark more convenient.

The `build.sh` script fetches the required software packages from the Internet using wget and Subversion. It then builds the registration tool, SIPp, and alog. After a successful build, the resulting binaries, that is `sip_reg`, `sipp`, and `alog`, are placed in the directory where the scripts was invoked. The script requires bash, wget, Subversion, GNU tar, C compiler, and GNU make.

[1] http://www.openser.org/
[2] http://sipp.sourceforge.net/

File	Description
README	instructions
alog.c	server log analysis
benchmark.sh	benchmarking script
build-sut.sh	SUT build script
build.sh	tools build script
calibrate.sh	benchmarking first stage
demo.cfg	OpenSER configuration
duac.xml	SIPp scenario
gencallgraph.octave	results formatting
gencallstats.octave	results formatting
iters.c	load generator
loadan.sh	server log analysis
loadmon.sh	log monitor script
ml.c	load generator
openser-1.1.0-notls_combined_ts-v3.patch	OpenSER patch
options.xml	SIPp ECHO scenario
register.sh	user registration
run-sut.sh	SUT run script
sip_reg.c	user registration
sippan.py	SIPp log analysis
vman.py	server log analysis

Table 1: Benchmark package contents.

The `build-sut.sh` script first fetches OpenSER sources from the Internet using wget. It then patches the sources with the patch included with the benchmark and builds OpenSER and installs it under `sut` directory to where the script was invoked from. The benchmark configuration (`demo.cfg`) for OpenSER is installed under the same directory hierarchy where OpenSER is installed to. The script patches (using sed) absolute paths into the OpenSER configuration file and thus the OpenSER install directory is not movable. The script also builds the load generator executables, `iters` and `ml`, and places them to the directory where the script was invoked from. The script requires bash, sed, wget, GNU tar, C compiler, and GNU make.

Both scripts have variables that can be configured by the user at the start of the script. There are C compiler and make variables in both scripts, and `build.sh` also includes the SIPp subversion revision to use.

2.2.1 Known problems

The various packages' build systems do not seem to tolerate paths with directories that have spaces in their names. The build scripts have some preparation for the problem, but are not well-tested in the case.

2.3 OpenSER

OpenSER is used as the main SUT software. The benchmark implementation instruments OpenSER to output time stamps of events that are needed for performance analysis. OpenSER is configured to output events to syslog so that its performance can be later analysed from the log. The version of OpenSER used by the benchmark is 1.1.0-notls.

2.3.1 Instrumentation

The OpenSER instrumentation is in the OpenSER patch file (see Tab. 1.) The patch inserts time stamps in log events from the accounting module. It also makes the registrar module log timing information (time stamp and lookup time) of all lookups. Time stamps are derived using the standard `gettimeofday()` library function and expressed in microseconds.

2.3.2 Configuration

The benchmark's default configuration uses the standard OpenSER configuration, but adds accounting and options modules. The thread and memory usage of OpenSER are made configurable by the benchmark scripts. The default is to use in-memory database for the user register.

The accounting module is needed for accounting the call events to the system log. By default the accounting module logs the INVITE answered, ACK received, and BYE answered events. For proper call length analysis, the configuration has logging added at the reception of INVITE in the routing script.

The options module is used for the ECHO test required by the benchmark specification. The SIP options request is basically answered statelessly, and as such it fits well as a transport test.

2.3.3 Known problems

Under some circumstances OpenSER log events are not correctly ordered in time. This is probably a bug in the accounting module and might be fixed by a later OpenSER version.

2.4 Load generator

The load generator consists of two C language programs, `iters` and `ml`. The main load generating algorithm has an unrolled loop that does summation and writing over a memory area. The `iters` program runs the loop for 1000000 rounds, times the execution, and finally calculates how many rounds the underlying hardware can execute the loop in 10 ms. The `ml` program then uses the results of `iters` and creates a user-specifiable load to the system by creating a new process every second that executes the loop for as many rounds as the specified load requires.

2.4.1 Known problems

The programs are not aware of nor tuned for SMP environments.

2.5 Load monitor

The load monitor is implemented as a simple bash script in `loadmon.sh`. It repeatedly runs `date` and `vmstat` and sleeps for one second. The idea is to create a log with the SUT's CPU and memory usage information with dates. The script is started by the `run-sut.sh` script along with OpenSER.

2.5.1 Known problems

The load monitor along with the load analysis script is tied to the Linux system, because it uses the `vmstat` program.

2.6 SIPp

SIPp is used as the user agents in the benchmark implementation. Two call scenario files are provided for SIPp by the benchmark implementation: `duac.xml` and `options.xml`. The user agent server uses the SIPp default UAS scenario. The used SIPp version is subversion revision 57, but it can be configured in the `build.sh` script.

The `duac.xml` scenario is a standard SIP user agent scenario. It send INVITE with user information from a database of users (created by the `register.sh` script), waits for the response, sends ACK, pauses for the call length, sends BYE, and waits for the response.

The `options.xml` scenario is for the client side of the ECHO test. It sends a SIP OPTIONS request and waits for the answer. SIPp's response time view can be used to see the ECHO latencies. The scenario has a response time partition tuned for modern hardware and low latency network. Under special high latency circumstances the partition should probably be changed.

2.6.1 Known problems

In `duac.xml`, a call always has the same caller and recipient, which might not reflect a real situation very well.

2.7 Benchmark scripts and tools

The benchmark implementation uses several scripts to run the benchmark. The bash script `register.sh` registers a number (default is 1000) of users to OpenSER using the `sip_reg` program and creates a user database csv file for SIPp. The `calibrate.sh` and `benchmark.sh` bash scripts are used to run the benchmark and the `run-sut.sh` script is used to start the SUT.

The user registration uses the `sip_reg` program, which is written in C. The program uses GNU libosip and libeXosip libraries for a SIP protocol implementation. The program sends SIP REGISTER messages to a server and waits for the responses. The user names have user specified name formatting.

The `calibrate.sh` script runs SIPp for a given number of seconds using the benchmark call scenario. Starting at a given calls per second rate, the script uses `sippan.py` to analyse the SIPp log after the run has ended. If the realised call rate corresponds to the one requested, the call rate is increased and the call scenario re-run. The cycle is repeated until a call rate that the server can barely sustain is found.

The `benchmark.sh` script performs the benchmark by running SIPp for a given number of seconds using the benchmark call scenario and a given call rate. After the benchmark the script prints the realised call rate, using `sippan.py` to analyse the SIPp results log.

2.7.1 Known problems

The `register.sh` script uses GNU seq command, which is not portable to all unix systems.

The `calibrate.sh` and `benchmark.sh` scripts are not SMP aware. SMP support would require running the load generator on every processor.

2.8 Log analysis tools

The benchmark implementation includes two tools for server log analysis and two octave scripts to produce graphs and statistics from the log analysis results. There is also one tool for analysing SIPp logs.

2.8.1 Server log analyser

The `alog` program is the primary server log analysis tool. It is written in C. The function of `alog` is to read and analyse the log that OpenSER writes to the system log, that is, the statements output by the accounting and registrar modules.

The program reads standard input line by line, optionally ignoring lines that do not fall inside a given span of dates. All log lines that are output by the "openser" process are further analysed, rest lines are skipped.

Log lines from openser are parsed and events with their timestamp are formed into singly linked lists. Separate lists are kept for lookups, incomplete calls, and complete calls.

Lookup events are added to a list in increasing time stamp order. Call events are first added to the incomplete calls list, and then, when all call events (INVITE, INVITE answered, ACK, BYE) are encountered, the call is moved to the complete calls list. Call events are not kept in timestamp order. A call may have multiple events of the same kind; for INVITE the latest INVITE before the answer to INVITE is kept, for other events, the first encountered event is kept.

After EOF, the lookup and call structures are analysed and optionally output to Octave or CSV files.

2.8.2 Server load log analyser

The server load log can be analysed using the `loadan.sh` bash script. The script uses sed to filter out log lines that do not fall into a given date interval. It then feeds the results to the `vman.py` Python script, which calculates and outputs CPU and memory usage statistics.

2.8.3 Graph and statistics output

The octave scripts `gencallgraph.octave` and `gencallstats.octave` can be used to output graphs and statistics from `alog` output. They are implemented using GNU Octave and also require gnuplot. The former script outputs a graph of calls per second, including completed calls, deadlined calls, and failed calls as separate curves. The latter script outputs statistics from call data that is more detailed than the statistics output by `alog`.

The `gencallgraph.octave` script reads a file (-cpu.octave) output by `alog` into a matrix and uses Octave's interface to gnuplot to plot the complete, deadlined, and failed calls to a single graph.

The `gencallstats.octave` reads a file (-calls.octave) output by `alog` into a matrix and prints timing values for all call events. Printed valued include minimum, maximum, mean, median, variance, and 10th, 25th, 50th, 75th, and 90th percentiles of lengths between events. The measured events are the full call from INITIATE to CLOSE, and all the intervals between two successive events. For failed calls, different event types are counted and printed out.

2.8.4 Known problems

The `alog` program is not written according to good programming principles.

The server load log analysis tools are tied to the Linux system, because they rely on the output format of `vmstat`.

3 Usage instructions

This section documents the usage of the benchmark and related tools. Running the benchmark can be outlined with the following steps:

1. Setting up the SUT.
2. Setting up the user agents.
3. Optionally, running the ECHO test.
4. Calibrating the call rate for the SUT.
5. Performing the benchmark.

3.1 Setting up

The benchmark is composed of one package that contains scripts for building and running the benchmark and tools. The package includes a README file with software dependencies that need to be met.

The network configuration should have 4 nodes. We label them, following the specification, CLIENT, SERVER1, SERVER2, and SUT. The nodes' IP addresses are referred to as CLIENTIP, SERVER1IP, SERVER2IP, and SUTIP, correspondingly.

3.1.1 Setting up the SUT

First, the SUT software needs to be built:

```
SUT> tar xzvf impl-yyyy-mm-dd.tar.gz
SUT> cd impl
SUT> ./build-sut.sh
```

Then, the load generator must be calibrated for the SUT machine:

```
SUT> ./iters
```

Finally, the SUT software can be started with:

```
SUT> ./run-sut.sh SUTIP threads memory CPUload CPUiters auxlogfile
```

Where *threads* is the number of threads OpenSER shall use (8 or more recommended), *memory* is the amount of memory needed for OpenSER (should be 32M for every 1,000 users or more), *CPUload* is the desired amount of background CPU load percentage (use 0 for no load), *CPUiters* is the number reported by iters in the previous step, or 0 if not needed, and *auxlogfile* is the file where memory and CPU load statistics are collected.

3.1.2 Setting up the user agents

First, the user agent servers need to be built:

```
SERVERn> tar xzvf impl-yyyy-mm-dd.tar.gz
SERVERn> cd impl
SERVERn> ./build.sh SUTIP
```

After building the SERVER nodes, the SIPp servers should be started:

```
SERVERn> ./sipp -sn uas
```

The same steps should be done on all of the SERVER nodes (SERVER1, SERVER2.)

After setting up the SERVER nodes, The CLIENT node software needs to be built:

```
CLIENT> tar xzvf impl-yyyy-mm-dd.tar.gz
CLIENT> cd impl
CLIENT> ./build.sh SUTIP
```

After building, the ECHO test can optionally be run:

```
CLIENT> ./sipp -sf options.xml SUTIP
```

The ECHO test latencies can be monitored using the SIPp latency view, which can be entered by pressing '3' in the SIPp terminal.

3.2 Running the benchmark

First, find out the CPS (Calls Per Second) rate the SUT is capable of handling with the supplied script calibrate.sh:

```
CLIENT> ./register.sh 2 SUTIP SERVER1IP SERVER2IP
CLIENT> ./calibrate.sh 900 start step SUTIP CLIENTIP
```

Replace *start* with a CPS rate the SUT can easily handle and *step* with approx. 2% of *start*. Larger values of *step* will run faster, but are less accurate.

After calibrating, the benchmark can be run:

```
CLIENT> ./benchmark.sh 900 CPSrate SUTIP CLIENTIP
```

Where *CPSrate* is the rate reported by the previous calibration step. It is also recommended to clear Syslog before benchmarking, so that the to be analyzed data is absolutely correct.

3.3 Result analysis

The server log can be analysed using the `alog` program. The basic usage is:

```
SUT> ./alog -o basename -b begin -e end < /var/log/messages
[or /var/log/syslog or some another used syslogging facility]
```

Where *begin* and *end* are the beginning and ending dates of the benchmark and *basename* is prefix for output filenames. Running alog will produce two files, called "*basename*-calls.octave" and "*basename*-cpu.octave". The "-calls" file contains the call length information and the "-cpu" file contains calls per second information in GNU Octave compatible table format.

The two files can then be processes by the Octave scripts to produce statistics and graphs:

```
SUT> ./gencallstats.octave basename-calls.octave
SUT> ./gencallgraphs.octave basename-cpu.octave output_terminal
```

Where *output_terminal* can be one of "x11", "postscript" or "aqua". The "aqua" terminal is native to Mac OS X, the others should work on most unix systems.

The SUT's CPU and memory usage statistics can be produced from the load monitor log by using:

```
SUT> ./loadan.sh auxlogfile begindate enddate
```

Where *begindate* and *enddate* should be in the form given by the SUT system's date command.

The `alog` program has more functionality than the previous example reveals. Following is a description of the program's command line switches:

-h Shows short usage instructions.

-q Suppresses runtime output of progress. Repeating the parameter suppresses output even more.

-l List details for failed calls. The contents of the failed calls' data structures are listed. This option is mainly useful for debugging, either alog or the benchmark.

-o *basename* Output lookups and calls to *basename*-calls, *basename*-cpu, and *basename*-lookups files. The format of the files is "Octave" by default, but can be changed to "CSV".

-u *time* Unit for calls-per-unit (-cpu) file in ms (default: 1000000 ms.)

-c Use CSV output format for the files.

-d Drop calls with time stamp order errors from output files. Sometimes OpenSER logs events in the wrong order and causes calls to have, for example, earlier closing time stamp than starting time stamp. These calls skew the statistics and should most likely be dropped from output files. The calls are still reported by `alog` terminal output.

-b *date* Begin date for log parsing (inclusive). All log entries that are earlier than *date* are skipped. The date option should be given as '-b "2006 Nov 1 15:47:25"'.

-e *date* End date for log parsing (inclusive). All log entries that are later than *date* are skipped.

The Octave and CSV output files contain fields of the various logged events. The general format for the Octave files is as follows:

```
<FIELD 1> <FIELD 2> <...> <FIELD n>
<FIELD 1> <FIELD 2> <...> <FIELD n>
...
```

The general format for the CSV files is as follows:

```
<FIELD 1>,<FIELD 2>,<...>,<FIELD n>
<FIELD 1>,<FIELD 2>,<...>,<FIELD n>
...
```

The Octave lookup files (”-lookups.octave”) contain the following fields:

FIELD 1 lookup time stamp

FIELD 2 lookup time

The Octave call files (”-calls.octave”) contain the following fields:

FIELD 1 starting time stamp of call

FIELD 2 a bitmask (as an integer) of received call events

FIELD 3 time stamp difference of ”bye replied” and ”invite received” events, which is effectively the length of the call

FIELD 4 time stamp difference of ”invite replied” and ”invite received” events

FIELD 5 time stamp difference of ”ack received” and ”invite replied” events

FIELD 6 time stamp difference of ”bye replied” and ”ack received” events

The Octave calls-per-unit files (”-cpu.octave”) contain the following fields:

FIELD 1 current relative time

FIELD 2 completed calls in this time unit until the next time unit

FIELD 3 deadlined calls in this time unit until the next time unit

FIELD 4 failed calls in this time unit until the next time unit

The CSV lookup files (”-lookups.csv”) contain the following fields:

FIELD 1 lookup time stamp

FIELD 2 looked up user’s name in double quotes

FIELD 3 lookup time

The CSV call files (”-calls.csv”) contain the following fields:

FIELD 1 starting time stamp of call

FIELD 2 call id in double quotes

FIELD 3 a bitmask (as a hexadecimal integer) of received call events

FIELD 4 time stamp difference of ”bye replied” and ”invite received” events, which is effectively the length of the call

FIELD 5 time stamp difference of ”invite replied” and ”invite received” events

FIELD 6 time stamp difference of ”ack received” and ”invite replied” events

FIELD 7 time stamp difference of ”bye replied” and ”ack received” events

The CSV calls-per-unit files (”-cpu.csv”) contain the following fields:

FIELD 1 current relative time

FIELD 2 completed calls in this time unit until the next time unit

FIELD 3 deadlined calls in this time unit until the next time unit

FIELD 4 failed calls in this time unit until the next time unit

All of the time stamps and time intervals are in microseconds. The bitmasks of call events contains the following bits, bit 0 being the least significant bit.

BIT 0 invite received

BIT 1 invite replied

BIT 2 ack received

BIT 3 bye replied

References

1 H. Lindholm and T. Vähäkangas. *Control plane benchmark specification*. Unpublished, 2006.