STOPOS

Willem Vermin, willem.vermin@surfsara.nl
Bas van der Vlies, bas.vandervlies@surfsara.nl

2013-02-01

Abstract

In this article we describe the internals and externals of the program stopos. Stopos is primarily designed to facilitate the running of many tasks on a cluster computer. It is based on managing lines in a pool. These lines will in general be used as command lines to define tasks. Stopos is heavily inspired by ToPoS (https://grid.sara.nl/wiki/index.php/Using_the_Grid/ToPoS) and experiences with disparm (https://www.surfsara.nl/systems/lisa/software/disparm). This project was carried out as a "seed-fund project" at SURFsara.

Introduction

A typical use of a cluster computer, such as SURFsara's Lisa system [1], is running many tasks each task with different parameters. A typical example is a parameter scan. A typical job allocates a node, and runs a number of tasks in parallel. The number of tasks should in general be equal to the amount of cores and/or the amount of memory. Sometimes, thousands of these jobs are necessary, and normally, the job scripts are generated by a script. In a homogeneous cluster (all nodes have the same amount of cores and the same amount of memory) this is clumsy but doable, but when the cluster is heterogeneous (such as the Lisa cluster), it becomes very hard to use all types of nodes available, and adapt the number of processes to the node where the job runs on. To deal with this problems, stopos was created: a server that manages a pool of lines. These lines are defined by the user, and the server serves the lines to jobs, such that each line is produced once. Earlier successful solutions are ToPoS[2] and disparm[3]. ToPoS is designed to be used on a server that can be accessed from the Grid, while disparm is based on files in the user's home directory. We think that ToPoS is not optimal for the Lisa system because of lack of cluster-based authentication and the universal access. The disparm approach is in these respects better, but in practice it appeared that the performance under heavy usage the performance was too low, and the reliability was not 100%: sometimes a line was produced more than once, or lines were not produced at all. Also, it is not possible to add lines dynamically: all lines have to be inserted at the beginning. We think that the approach followed in stopos is a well balanced solution, particularly fit to be used in a cluster environment.

Usage

A typical usage of stopos is:

- fill the pool with lines

[1] https://www.surfsara.nl/systems/lisa

[2] https://grid.sara.nl/wiki/index.php/Using_the_Grid/ToPoS

[3] https://www.surfsara.nl/systems/lisa/software/disparm

- start jobs, each job gets one or more lines from the pool, and remove the lines that are not needed any more[4]
- maybe: add more lines to the pool

Stopos is implemented as a shell function in the bash shell, or as an alias in the (t)csh shell. Stopos calls the program stoposclient and transforms it's output to environment variables, the most important being STOPOS_RC and STOPOS_VALUE.
The commands to create the pool:

```
 module load stopos       # activate the stopos software
 stopos create            # create the pool
 stopos add parmfile      # add the lines in the file parmfile to the pool
```

In a job script, one could use code like this:

```
module load stopos           # activate the stopos software
 stopos next                  # get a line from the pool
 if [ $STOPOS_RC = OK] ; then # check the return code
   run_my_prog $STOPOS_VALUE  # run the program using the line as parameter
 fi
```

A more elaborate job script, using all cores available on a node is available in the Appendix, together with the man page of stopos.

Tasks and communication

Stopos is a shell function or an alias that calls stoposclient. Stoposclient communicates with stoposserver via an http server (apache[5], thttpd[6], ...). The server sends commands to the server, and transforms the answers to a format that are sourced by stopos to define environment variables. Care is taken that communications between the client and the server are short in length: at most a few kilobytes are sent between client and server.

Tasks of the client

The client parses it's command line, sends the appropriate command to the server and receives the response. This response is put in a form that can be 'eval-ed' so that environment variables are set.

The client recognizes the following commands on its command line:

- `-h,--h` : `prints a short usage message`
- `-v,--v` : `prints the version number`
- `create [ -p,--pool POOL ]` : create a pool
- `status [ -p,--pool POOL ]` : get status from a pool

[4] Normally, a line is produced only once. However, the user can request that a line is produced and used again, for example in the case the previous task handling this line ran into a time limit.

[5] http://httpd.apache.org/

[6] http://opensource.dyc.edu/sthttpd

- `purge [ -p,--pool POOL]` : remove a pool
- `pools` : what are my pools
- `add [ -p,--pool POOL] [ FILENAME ]`: add a file to the pool
- `next [-m,--multi]` : get next line from the pool
- `remove [ KEY ]` : remove line
- `dump [ -p,--p POOL ]` : dump a line from the pool

For example:

```
stopos create -p mypool1
stopos add -p mypool1 parmfile
```

The -p (or –pool) flag defines the name of the pool file to be used ("parmfile" in this example). If no -p or –pool flag is present, the value of the pool is taken from the environment variable STOPOS_POOL. If that is not available, the value "pool" is used.
The -m (or –multi) flag signifies that lines that are already produces, may be produced again.
The default for FILENAME is standard input.
If no value is given for KEY with the "remove" command, the value of the environment variable STOPOS_KEY is used. This value is set by the "next" command.

The following environment variables are set by the command "stopos":

- `STOPOS_RC` : return code of the command. The value "OK" means that things went well
- `STOPOS_VALUE` : result of the "next" and "pools" command
- `STOPOS_COMMITTED` : number of times the "next" line was produced
- `STOPOS_KEY` : set by the "next" command. Used by the "remove" command
- `STOPOS_COUNT` : set by "status": number of lines added to the pool
- `STOPOS_PRESENT` : set by "status": number of lines in the pool
- `STOPOS_PRESENT0` : set by "status": number of lines never been committed

Another task is creating a unique id for the user. This id, combined with the login name of the user and the name of the pool, will be used by the server to identify the pool to be used. The id is stored in the HOME directory of the user, in the file $HOME/.stopos/rc. If this file is missing, or inappropriate, the client will create this file.

All commands result in a single communication with the server, except the "add" command: the client will send a number of "add" commands to the server, one for each line of the file.

The "dump" command is kind of special. It is created for the case a user wants to inspect the actual contents of the pool. The "dump" command will produce one line per call, and does not interfere with the other commands. (More specific: the "next" command and the "dump" command use different pointers) When the last line has been produced, the next call will result in an error code (STOPOS_RC != OK). Then the next call will start again at the first available record in the pool.

Messages from client to server

A message from client to server consists of the following fields:

- `stopos` : the string “stopos”
- `prot` : the protocol to be used (i.e. which kind of database)
- `id` : the identification of the user
- `pool` : the name of the pool to be used
- `command` : the command for the server
- `multi` : the value of “multi”: can a line be produced more than once?
- `value` : a parameter

All fields are coded in hexadecimal strings, embraced by a fixed header and trailer. This is to ensure that no problematic things (spaces, tabs, slashes, empty strings etc.) have to be transmitted. The fields are separated with the character “-”. Example: to send a message to the server located at “http://www.stoposserver.surfsara.nl” with:

- protocol = “gdbm”
- user identification = “willem.qR7w3XQp”
- poolname = “mypool”
- command = “add”
- multi = “no”
- parameter = “1 2 3”

The client would create the following url:

```
http://www.stoposserver.surfsara.nl?ZZZ73746f706f73YY  “stopos”
                                     -ZZZ6764626dYY     “gdbm”
               -ZZZ77696c6c656d2e7152377733585170YY     “willem.qR7w3XQp”
                                 -ZZZ6d79706f6f6cYY     “mypool”
                                       -ZZZ616464YY     “add”
                                         -ZZZ6e6fYY     “no”
                                   -ZZZ3120322033YY     “1 2 3”
```

The header is “ZZZ”, the trailer is “YY”.

Tasks for the server

The server parses the message from the client, executes the command and sends the results back. The commands the the server recognizes are:

- `create` : create a pool
- `status` : send the status of the pool
- `purge` : remove the pool
- `add` : add a line, taken from the "value" field of the message
- `next` : send the next line together with the "key" and "committed". Take care of "multi"
- `remove` : remove a line with key given in the "value" field
- `dump` : return next line to be dumped, together with "key" and "committed"
- `pools` : return the names of the available pools for the user

Not accidentally: these commands have the same names as the commands for the client, but the context is different.

Communication from server to client

The server parses the message from the client and sends the result back, using "Content-type: text/plain". The server puts the following string in the output:

```
STOPOS:
```

This string is immediately followed by the desired output, which consists of the following fields:

- `return message` : "OK" or some error message
- `key` : key of the record just retrieved or empty
- `committed` : number of times the record has been committed or empty
- `count` : number of records added so far or empty
- `present` : number of records present or empty
- `present0` : number of records never been committed or empty
- `value` : return value or empty
- `new line character`

The client will search for the string "STOPOS:" and use the string immediately following as the result. The developer can, if she wishes, print extra information and have a look at the total output of the server, as long as she does not use the string "STOPOS:". The fields are encoded, just like is done with the message from the client to the server, only the separator here is "+" for no special reason.

Internals of the client

The client parses the command line, using getopt(3). It performs some basic checks on the parameters given, and sends an appropriate command to the server, using libcurl(3). The output from the server is parsed and transformed into statements that can be executed in the shell. These commands set environment variables, with names starting with “STOPOS_”.

Internals of the server

The server parses the command sent from the client and an http-server (apache, thttpd), executes it and sends the results back to the client. Most of the interesting work is done in the class “stopos_pool”. This class, from which a database-specific class has to be derived, maintains a double linked list of lines. The addressing of the lines is based on keys that are computed from the counter that is incremented each time a line is added. The derived class has to deal with some basic I/O operations: creating, opening, closing and removing of the database; reading, writing and removing records. In this code, four database implementations are provided:

- based on gdbm [7]
- based om mysql [8]
- based on a flat ascii file
- based on files in a folder, each file containing one line

The client determines which implementation the server has to use. Storage of a record is based on the key, but, if desirable, the derived class can provide a suitable slot for storing the record. This feature is used in the “flat file” solution, in order to be able to re-use space occupied by removed records.

Storage of lines

Lines are stored in records, which also contain extra fields. The records are hexadecimal coded, just like the communication from client to server. The separation character is here ’/’. Records are stored using unique keys, generated by the class “stopos_pool”. The derived class has no knowledge about the meaning of the records: it’s only task is to store, retrieve and delete them.

Performance

A program “test_pool.cpp” has been provided which runs all four implementations, without an http server, but directly calling the functions from the class “stopos_pool” and reports how much time a implementation takes. Furthermore, it does some consistency checks. The pool consists of a few hundreds of lines. The results are[9]:

```
database   time (sec)
gdbm        3.31
flatfile    0.19
files       0.73
```

[7] http://www.gnu.org.ua/software/gdbm/

[8] http://www.mysql.com/

[9] The program was run on a Linux workstation, equipped with a “normal” (no SSD) disk

mysql	4.71

The winner is clearly the “flat file” implementation. Disadvantage of this solution (and also of the “files” and “mysql” solution) is the amount of space that is used for the database: each line takes 4 Kbyte. The “gdbm” solution does not have this problem.
Normally, the stopos software will be used using an http server, so it is more relevant to have look at the timings in this case. As an example, that man page of bash (5459 lines) was put in the pool, and all lines were retrieved and removed:

database	time(sec)
gdbm	187
flatfile	182
files	190
mysql	309

These timings show, that for performance reasons, there is not much difference between “gdbm”, “flatfile” and “files”. We choose the “gdbm” implementation, because of it’s economy with disk space used.

Installation

The source tree contains a file “Make.inc”, defining variables that determine the version of the program, the installation directories and which database versions should be produced. This file will be included in the Makefile. A trivial script “makeit” sets the umask and calls make(1) to create and install the executables. The script “clean” executes a “make clean” to get rid of compilation products.

Extra tools

The source tree also contains the scripts “sara-get-num-cores” and “sara-get-mem-size”, along with their man pages. Sara-get-num-cores prints the number of available cores, while sara-get-mem-size prints the amount of memory. Both numbers can be used to determine how many processes should run in parallel on a given node. These scripts and man pages will be installed also by calling “./makeit”.

Example of a job using stopos

```
#PBS -lnodes=1 -lwalltime=3:00:00
module load stopos
ncores=`sara-get-num-cores`
for ((i=1; i<=ncores; i++)) ; do
(
  for ((j=1; j<=10; j++)) ; do
    stopos next -p pool1
      if [ "$STOPOS_RC" != "OK" ]; then
      break
     fi
     eval "scanner $STOPOS_VALUE"
     stopos remove -p pool1
13      done
14    ) &
15  done
16  wait
```

- 1 Note: we do not specify how many cores a node should have.
- 2 Load the stopos module.
- 3 Determine the number of cores available on this node: Module stopos contains 'sara-get-num-cores': this command prints the number of cores available on the node. The command 'sara-get-mem-size' prints the available amount of memory in Mbytes. We do not use it here, but this number could be interesting when running memory intensive programs.
- 4 Loop over the number of cores. The content of the loop is placed in the background (see 14)
- 6 Loop over the number of the desired number of repetitions.
- 7 Here we get the value of the next line, delivered in the environment variable STOPOS_VALUE. The environment variable STOPOS_RC has the value "OK" if everything went ok. STOPOS_KEY contains a value that can be used to remove the line from the pool. We use the pool 'pool1', created before.
- 8 .. 10 Break out the loop if stopos fails (usually because there are no lines left).
- 11 Use the parameters in $STOPOS_VALUE. The 'eval' and quotes ensure that spaces and quotes in the parameters are correctly interpreted.
- 12 Remove the line we just processed from the pool.
- 14 Notice the &: this takes care that ncores instantiations of scanner will be started in parallel.
- 16 The 'wait' ensures that the job script will only finish after all background processes are ended.
- The job can be submitted a number of times with one 'qsub' command, for example: `qsub -t 1-100 job`

Man page of stopos

STOPOS(1) STOPOS(1)

NAME

stopos − alias to call the program stoposclient and eval it's output

SYNOPSIS

```
       stopos  -h,--help

       stopos  -v,--version

       stopos  create    [ -p,--pool POOL ]

       stopos  status    [ -p,--pool POOL]

       stopos  purge     [ -p,--pool POOL ]

       stopos  pools

       stopos  add       [ -p,--pool POOL ] [ FILENAME ]

       stopos  next      [ -p,--pool POOL ] [ -m,--multi ]

       stopos  remove    [ -p,--pool POOL] [ KEY  ]

       stopos  dump      [ -p,--pool POOL]

       The -q,--quiet flag supresses most output.

DESCRIPTION
       Stopos  is an alias using the program stoposclient , an utility to store
       and retrieve text lines in a pool.  In general, the text lines are used
       as command parameters, see Examples.

COMMANDS
       -h,--help
           Prints usage information.

       -v,--version
           Prints version.

       create [ -p,--pool POOL ] [ FILENAME ]
           Creates new pool.  A pool with the same name will be removed first.

       status
           Prints to standard error        environment variable
          -------------------------------------------------------
           total number of lines added       STOPOS_COUNT
           number of lines present           STOPOS_PRESENT
           number of lines never committed   STOPOS_PRESENT0


       purge [-p,--pool POOL ]
           Removes the pool.

       pools [-p,--pool POOL ]
```

Lists on standard error the pools available. The environment variable STOPOS_VALUE is set accordingly.

add [−p,−−pool POOL] [FILENAME]

Adds lines from FILENAME, default from stdin. The environment variables STOPOS_RC and STOPS_KEY are set.
NOTE: when stopos is reading from a pipe as in:

```
cat parmfile | stopos add
```

no environment variables are set.

next [−p,−−pool POOL] [−m,−−multi]

Gets the next line from the pool. The environment variables STOPOS_VALUE, STOPOS_COMMITTED and STOPOS_KEY are set, see ENVIRONMENT. By default, the same line in the pool will be produced only once. When all lines are commited, STOPS_RC will not be equal to OK. Using the −−multi flag, the same line can be produced more than once, if necessary stopos will wrap around. This can be useful for dealing with crashed jobs.

remove [−p,−−pool POOL] [KEY]

Removes the line with key KEY as ready. If not specified on the commandline, the value of environment variable STOPOS_KEY is used.

dump [−p,−−pool POOL]

Reads the next available line, and puts it, preceded by it's key and number of commitments in environment variable STOPOS_VALUE. The first call will produce the first line. When all lines have been delivered, STOPOS_RC gets a value other than OK. A subsequent call will start the dump with the first line again.

OPTIONS

−p,−−pool POOL

POOL is a unique name of the pool. Default: pool. The name of the pool can also be set using the environment variable STOPOS_POOL. The command line flag has precedence.

ENVIRONMENT

Stopos sets the following environment variables:

STOPOS_RC if the value is OK, than no errors were found

STOPOS_KEY contains the keyvalue of the line produced with the 'next' command

STOPOS_COMMITTED contains the number of times the line has been committed

STOPOS_VALUE contains the line produced by the 'next' command or the result of the 'pools' command

STOPOS_COUNT STOPOS_PRESENT STOPOS_PRESENT0 see the 'status' command above

Stopos uses the following environment variables:

STOPOS_POOL see the description of the --pool flag under OPTIONS

STOPOS_KEY see the 'remove' flag above

STOPOS_SERVER_URL the url stopos uses to access the server, default http://stopos.osd.surfsara.nl/cgi-bin/stoposserver

EXAMPLES

Create a pool with the first 10 lines from the man page of sed:

```
man sed | head > parmfile
stopos create
stopos add parmfile
```

Get a line:

```
stopos next
```

The environment variable STOPOS_VALUE contains now one of the first ten lines of the man page of sed.

You can use this in a command like this:

```
eval "myparser $STOPOS_VALUE"
```

The line can be removed by:

```
stopos remove
```

Finally, the pool can be completely removed by:

```
stopos purge
```

FILES

$HOME/.stopos/id

This file is set by stopos to store an unique id, which, together with the login name, will be used to identify which pools are yours.

SEE ALSO

sara-get-mem-size(1), sara-get-num-cores(1), stoposclient(1), stoposdump(1)

AUTHORS

Willem Vermin

BUGS

No bugs known yet.