Exploring improvements to the LHCb ELOG electronic logbook

Public Note

Issue:	1
Revision:	0
Reference:	LHCb-PUB-2015-020
Created:	August 13, 2015
Last modified:	September 1, 2015
Prepared by:	Philip Leung, Niko Neufeld

Abstract

ELOG, the electronic logbook used by LHCb, has been suffering from poor accessibility and slow performance. Investigating revealed that a combination of inefficient searches, caused mainly by frequent reads of files stored in NFS, and an inability for the server to handle concurrent operations were rendering the service unusable for up to a minute when users performed search or sort operations.

By adding a minor patch to the ELOG source code, moving data to local storage and optimizing server configuration the project was able to reduce search-times for the largest of the logbooks being used at LHCb to 30% while also improving possibilities for future growth by allowing for concurrent use and accelerating the most common search-operations in a way which should stay consistent over extended time-periods.

Document Status Sheet

1. Document Title: Exploring improvements to the LHCb ELOG electronic logbook			
2. Document Reference Number: LHCb-PUB-2015-020			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Draft	1	August 21, 2015	First version
Draft	2	August 28, 2015	Finished draft but don't yet have ref number
Final	1	September 1, 2015	Added ref number

Contents

1 Introduction . . . 3
2 Background . . . 3
3 Problem . . . 3
4 Considerations . . . 3
5 Testing . . . 4
5.1 Search-times . . . 4
5.2 Machine Resource Usage . . . 4
5.3 Code Profiling . . . 5
5.4 Network I/O . . . 5
5.5 File-system I/O . . . 5
6 Improvements . . . 5
7 Future work . . . 6
8 Conclusion . . . 7
9 References . . . 7
Appendix A el_retrieve function . . . 8

Appendix B Implementing ELOG multi-instance support 12

List of Tables

1 Search-time test results in milliseconds 4
2 Search-time test results in milliseconds with local storage 6
3 Search-time test results in milliseconds with local storage and multi-instance support 6

1 Introduction

An ELOG[1] electronic logbook is currently being used by LHCb mainly to provide logging operations for the different sub-detector groups under LHCb and to act as a shift log. It has been in use since year 2006, currently has over 800 registered users and is now showing signs of poor scalability which can significantly slow down usage. This note will investigate the causes of bad performance and possible improvements to the current system.

2 Background

ELOG was developed in the C programming language by the physicist Stefan Ritt at the Paul Sherrer Institut (PSI) in 2001 as a tool to replace paper logbooks and desktop electronic logbook clients It is an open source web-application which runs single-threaded and is backed by a database of plain-text ASCII files. The software is highly configurable using configuration scripts and CSS-files to allow even less technically experienced users to quickly and easily set up working logbooks with a decent range of functionality. Users post entries either through the web interface or by using the ELOG command-line interface.

Efforts were made by the creator to keep dependencies to a minimum which is why it uses limited third party libraries and even contains its own integrated web server. In LHCb the logbook is also run behind an Apache proxy for redirection and caching purposes.

3 Problem

Being developed for relatively small projects ELOG has not had a need to scale particularly well. However, in a large experiment like LHCb this becomes a important issue which can significantly impact user productivity.

Although the system works well when there is limited data and users to handle the single-threaded nature and inefficient logbook searches make it so that users can be locked out of performing even very simple tasks.

Long searches will hog the single thread of the ELOG instance, keeping it from responding to other requests until it is finished. When this happens it gives a bad user experience which has lead to many users believing that the service has failed when it is simply busy. This slows down the work of users and puts unnecessary burden of investigating tickets on the ELOG administrators. Because it is not possible to interact with the system while it is performing an action it means that if a user starts a long search and changes their mind it is impossible to cancel and everybody has to wait for it to finish before they can continue using the service.

Due to the static and stateless nature of the software ELOG handles each displayed page as a completely separate request - performing a search when changing pages in search results, leading to a clunky user experience, loss of productivity and a more lack of responsiveness for the system. This can cause much more downtime as changing pages takes as much time as a completely new search.

4 Considerations

Because the ELOG system is already in use and is to be maintained for an unforeseen amount of time some considerations had to be taken when considering improvements to the system;

- Minimal changes to source code is greatly preferred as this would have to be re-applied for every ELOG update, would be likely to have adverse effects in the future, and would require coming administrators to study the code and learn the compilation-process. This is especially inadvisable as the code is quite bulky due in part to the lack of dependencies and contains functions exceeding 2000 lines with very few comments.

- ELOG allows for much customization and has many accelerators as well as alternate input-methods that users have spent time configuring and getting accustomed to. For this reason it is of significant value to keep the customizations and means of interaction working so as to avoid loss of productivity and the possibility of having to update automated systems.
- The logbook server is under constant use so very frequent changes to the production instance or actions occupying the single thread for too long would hurt users. Because of this testing and development would be best done on a substitute whenever possible.

5 Testing

In order to determine the cause of slow search performance a number of tests were run. These tests were carried out on virtual machines running Scientific Linux 6.5 just like the production server.

5.1 Search-times

Time taken from the user's perspective to perform search operations on logbooks copied from the active lblogbook server was measured using JMeter test-benches. Because ELOG is single-threaded only one test-thread could be run at a time. Additionally, as the ELOG-instance is under constant use these test-cases were run on a substitute machine with copies of the ELOG-server configuration and logbooks.

A three-hundred tests in which free-text searches for posts made by a certain author were triggered over http were run for each of the four logbooks of different sizes.

		Time (ms)			
Logbook size (# Entries)	**# Samples**	**Average**	**Min**	**Max**	$\frac{\textbf{Average time (ms)}}{\textbf{\# Entries}}$
Tiny (1)	300	10	7	34	10.000
Small (464)	300	187	159	320	0.403
Medium (11332)	300	4573	3812	7220	0.404
Large (76270)	300	47767	44357	69353	0.707

Table 1 Search-time test results in milliseconds

As is shown in Table 1 search-times grow quite quickly from barely noticeable to very slow. It is also noted that there seem to be outliers going towards slower times as the average is usually much closer to the minimum. The possible reasons for this are numerous as tests were run on a shared VM which accesses files over the network.

5.2 Machine Resource Usage

CPU and memory usage were monitored using the Linux htop and sar commands. This revealed some surprising behaviour in the program; Firstly, when a search is triggered the CPU usage of the program rises very slowly and was only able to reach maximum usage of available capacity in longer searches which allow it over 40 seconds to build up. The reason for this could not be found but it is suspected that it is connected to the search algorithm used or that it is waiting for something like file I/O.

Secondly, ELOG's memory usage stays constant regardless of load on both the testing vm and on the production one. This indicates that the program may not be performing optimally and that maybe it was limiting itself to an arbitrary amount of memory. The reason for this was not found in the code and the problem is inconsistent with a Windows system tested which increased memory usage under load as expected.

As a result of these findings it was concluded that improving search performance simply by increasing the amount of available random access memory would not be likely to help. As for upgrading the CPU; the single-threaded nature of the application means that it would require a very fast single core which may not even show as much improvement as expected due to the strange behaviour exhibited.

5.3 Code Profiling

The code was profiled by downloading the version 3.1.1 ELOG source code from the official website[1] and compiling it in gcc using the -gp debugging flag by adding it to the existing flags used in the provided makefile and running the make command on the folder. Compiling with this flag causes a file to be generated after the compiled program has been run and successfully shut down which can then be analysed by the gprof profiling tool to extract information about which functions in the code were called how many times. Gprof can also measure the time spent in longer-running functions but this functionality did not register when running the ELOG-server in daemon-mode.

To compensate for this without having take down and run the production server as a regular application the same code was run separately with generic searches to estimate how long the different functions were ran. The result of this was that the function taking the most total time by far with 95.9% of active time taken was one called ***el_retrieve***, the source code of which is provided in appendix A, which is responsible for retrieving an ELOG entry from its id. A reasonable suspect for the time taken by this function is the use of the open(), a function call for opening files, which is not profiled by gprof as it is a standard library function. These results point towards there likely being room for improvement in I/O.

5.4 Network I/O

A round-trip time of less than one millisecond for Meyrin-site client-generated pings to the server with no packet loss indicates that the network connection seems very strong for the test-client. Using Wireshark on the client-side machine to investigate traffic to and from the server showed that the time taken by the server from sending an ACK for a large search request to beginning to return data dwarves any latency in client-server network communication. From these findings it does not seem likely that slow client-server network I/O is at fault for the slow searches.

5.5 File-system I/O

The logbook database is stored as plain text files on a mounted NFS-drive. Wireshark and nfsiostat reveals that communication from ELOG-server to NFS mostly takes under one millisecond, meaning that there is no problem with the connection between ELOG- and NFS-servers. There are, however, what might seem to be a quite large amount of traffic between the servers during search. Measuring the number of packages sent between the servers with Wireshark shows that in two minutes of running ELOG searches 316656 NFS packets were exchanged compared to the 28 NFS packages sent when the logbook server was idle for the same period of time. The average time between sending a NFS call and getting a reply was calculated to be about 335 microseconds though these times range from 7 microseconds GETATTR replies to 37 millisecond READ replies. There was also some RPC retransmissions of GETATTR calls which can be a sign of a slow NFS-server or of congestion which may be due to all the calls made by the logbook server.

A second search-time test was done placing the logbooks and binary files on a local disk of the testing VM which showed a clear improvement in performance. As can be seen when comparing the numbers presented in Table 1 to the new ones from Table 2 the average search-times for the large logbook plummeted from 47767 to 11951 ms which is an almost 75% reduction. It should be noted, however, that because of size restraints on the local disk used only the actual logbook being tested was able to fit. This will have an impact on logbook indexing speeds but should not affect the time taken to perform searches in singular logbooks.

6 Improvements

In order to handle multiple users at once a patch has been made to the source code which makes it possible to designate an ELOG instance as a search slave by inserting the line 'Search slave = 1' in the global scope of the .cfg file used. These are meant to be run in read-only mode and allows

		Time (ms)			
Logbook size (# Entries)	**# Samples**	**Average**	**Min**	**Max**	**Average time (ms) / # Entries**
Tiny (1)	300	8	6	30	8.000
Small (464)	300	22	20	44	0.047
Medium (11332)	300	548	319	884	0.048
Large (76270)	300	11951	9834	14512	0.157

Table 2 Search-time test results in milliseconds with local storage

multiple ELOG instances to be run on different ports while using the same logbooks so long as only a maximum of one instance has write permissions and is not designated a search slave. As the name suggests the search slaves are meant to offload long search and sort functions from the master instance which handles lighter operations from multiple users. A suggested way of going about this would be to have a proxy such as Apache redirect and distribute all request URLs pertaining to searching or sorting among the slave instances and let all other operations go through to the master instance. Steps taken to enable this functionality are documented in appendix B.

		Time (ms)			
Logbook size (# Entries)	**# Samples**	**Average**	**Min**	**Max**	**Average time (ms) / # Entries**
Tiny (1)	300	13	8	223	13.000
Small (464)	300	31	27	87	0.067
Medium (11332)	300	717	395	1465	0.063
Large (76270)	300	13939	10405	20321	0.183

Table 3 Search-time test results in milliseconds with local storage and multi-instance support

Another search-time test was run a search using the multi-instance solution while running on local disk. The results presented in Table 3 show a slight performance decrease compared to Table 2 but the long search still only takes about 30% of the time taken in the current system. It should be noted, however, that since only the logbooks that were being tested could fit on the local disk there will likely be some decrease in performance when running this solution on the entirety of the LHCb logbooks.

Beyond improving file access and allowing for concurrent users some analysis of Apache and ELOG access logs showed that many longer searches were being done where shorter ones would've sufficed. Limiting the default date-range for searches to posts made within past month if no other range is set can substantially improve search-times in long-running logbooks. This is done by adding the line 'Show last default = 31' to the cfg file for the logbook in question.

Finally, it was deemed that there was little to be done in terms of cost-effective hardware scaling other than giving the virtual machine a preference for hypervisors with faster CPU cores. Should demand for better performance rise one may also exchange the current storage of the more popular logbooks for faster SSDs or use more intelligent caching systems.

7 Future work

For future possible improvements to the system one which would have an immediate positive impact for LHCb lblogbook users would be the implementation of true multi-threading for singular instances of ELOG servers. This would require extensive changes in and understanding of the source code but would allow many users to access the logbook simultaneously without affecting each other. Judging from inspection of Apache access logs simply being able to run three simultaneous threads should be enough to adequately serve the current load of users in most cases. Luckily Stefan Ritt has noted that this issue is a high priority for him in the updates of ELOG and gave an expected time-frame of "months" from August 2015 to have a multi-threaded server ready for use.

Another improvement which could help productivity as the logbooks grow would be replacing the current search algorithms. Insufficient tests have been done for different number of logbooks entries to accurately say what level of time-complexity applies to the current algorithms used but as the numbers suggest that it is worse than O(N) there may be cause for changing he algorithms for future scaling.

Finally, implementing a cache or other accelerator for flipping between pages in search results locally without having to trigger a separate search should help those who are not able to find what they are looking for in the first page of results.

8 Conclusion

The project was able to reduce search-times for the largest of the logbooks being used at LHCb to 30% while also improving possibilities for future growth by allowing for concurrent use and accelerating the most common search-operations in a way which should stay consistent over extended time-periods. These improvements in conjunction with the eventual possibility of native multi-threading being implemented will hopefully allow the lblogbook to sufficiently serve its purpose in future LHCb research.

9 References

[1] Stefan Ritt, The ELOG Home Page, [Online] Available: `https://midas.psi.ch/elog/`

[2] Free Software Foundation, ”GNU gprof”, [Online] Available: `https://sourceware.org/binutils/docs/gprof/`

Appendix A el_retrieve function

```
/********************************************************************

  Name:      elogd.c
  Created by: Stefan Ritt
  Copyright 2000 + Stefan Ritt

  ELOG is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  ELOG is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
  GNU General Public License for more details.

  In addition, as a special exception, the copyright holders give
  permission to link the code of portions of this program with the
  OpenSSL library under certain conditions as described in each
  individual source file, and distribute linked combinations
  including the two.
  You must obey the GNU General Public License in all respects
  for all of the code used other than OpenSSL. If you modify
  file(s) with this exception, you may extend this exception to your
  version of the file(s), but you are not obligated to do so. If you
  do not wish to do so, delete this exception statement from your
  version. If you delete this exception statement from all source
  files in the program, then also delete it here.

  You should have received a copy of the GNU General Public License
  along with ELOG. If not, see <http://www.gnu.org/licenses/>.


  Contents:  Web server program for Electronic Logbook ELOG

\********************************************************************/

int el_retrieve(LOGBOOK * lbs, int message_id, char *date, char
    attr_list[MAX_N_ATTR][NAME_LENGTH],
            char attrib[MAX_N_ATTR][NAME_LENGTH], int n_attr, char *text, int
                *textsize,
            char *in_reply_to, char *reply_to, char
                attachment[MAX_ATTACHMENTS][MAX_PATH_LENGTH],
            char *encoding, char *locked_by, char *draft)
/********************************************************************
 Routine: el_retrieve

 Purpose: Retrieve an ELog entry by its message tab

 Input:
 LOGBOOK lbs        Logbook structure
 int   message_id   Message ID to retrieve
 int   *size        Size of text buffer

 Output:
 char  *tag         tag of retrieved message
 char  *date        Date/time of message recording
 char  attr_list    Names of attributes
 char  attrib       Values of attributes
```

```
    int   n_attr       Number of attributes
    char *text         Message text
    char *in_reply_to Original message if this one is a reply
    char *reply_to     Replies for current message
    char *attachment[] File attachments
    char *encoding     Encoding of message
    char *locked_by    User/Host if locked for editing
    char *draft        User who drafted that message
    int   *size        Actual message text size

    Function value:
    EL_SUCCESS         Successful completion
    EL_EMPTY           Logbook is empty
    EL_NO_MSG          Message doesn't exist
    EL_FILE_ERROR      Internal error

    \********************************************************************/
{
  int i, index, size, fh;
  char str[NAME_LENGTH], file_name[256], *p;
  char *message, attachment_all[64 * MAX_ATTACHMENTS];

  if (message_id == 0)
    /* open most recent message */
    message_id = el_search_message(lbs, EL_LAST, 0, FALSE);

  if (message_id == 0)
    return EL_EMPTY;

  for (index = 0; index < *lbs->n_el_index; index++)
    if (lbs->el_index[index].message_id == message_id)
      break;

  if (index == *lbs->n_el_index)
    return EL_NO_MSG;

  sprintf(file_name, "%s%s%s", lbs->data_dir, lbs->el_index[index].subdir,
      lbs->el_index[index].file_name);
  fh = open(file_name, O_RDONLY | O_BINARY, 0644);
  if (fh < 0) {
    /* file might have been deleted, rebuild index */
    el_build_index(lbs, TRUE);
    return el_retrieve(lbs, message_id, date, attr_list, attrib, n_attr, text,
        textsize, in_reply_to,
                    reply_to, attachment, encoding, locked_by, draft);
  }

  message = xmalloc(TEXT_SIZE + 1000);

  lseek(fh, lbs->el_index[index].offset, SEEK_SET);
  i = my_read(fh, message, TEXT_SIZE + 1000 - 1);
  if (i <= 0) {
    xfree(message);
    close(fh);
    return EL_FILE_ERROR;
  }

  message[i] = 0;
  close(fh);

  if (strncmp(message, "$@MID@$:", 8) != 0) {
    xfree(message);
```

```
      /* file might have been edited, rebuild index */
      el_build_index(lbs, TRUE);
      return el_retrieve(lbs, message_id, date, attr_list, attrib, n_attr, text,
          textsize, in_reply_to,
                       reply_to, attachment, encoding, locked_by, draft);
   }

   /* check for correct ID */
   if (atoi(message + 8) != message_id) {
      xfree(message);
      return EL_FILE_ERROR;
   }

   /* decode message size */
   p = strstr(message + 8, "$@MID@$:");
   if (p == NULL)
      size = strlen(message);
   else
      size = p - message;

   message[size] = 0;

   /* decode message */
   if (date)
      el_decode(message, "Date: ", date, 80);
   if (reply_to)
      el_decode_intlist(message, "Reply to: ", reply_to, MAX_REPLY_TO * 10);
   if (in_reply_to)
      el_decode_int(message, "In reply to: ", in_reply_to, 80);

   if (n_attr == -1) {
      /* derive attribute names from message */
      for (i = 0;; i++) {
         el_enum_attr(message, i, attr_list[i], attrib[i]);
         if (!attr_list[i][0])
            break;
      }
      n_attr = i;

   } else {
      if (attrib)
         for (i = 0; i < n_attr; i++) {
            sprintf(str, "%s: ", attr_list[i]);
            el_decode(message, str, attrib[i], NAME_LENGTH);
         }
   }

   el_decode(message, "Attachment: ", attachment_all, sizeof(attachment_all));
   if (encoding)
      el_decode(message, "Encoding: ", encoding, 80);

   if (attachment) {
      /* break apart attachements */
      for (i = 0; i < MAX_ATTACHMENTS; i++)
         if (attachment[i] != NULL)
            attachment[i][0] = 0;

      for (i = 0; i < MAX_ATTACHMENTS; i++) {
         if (attachment[i] != NULL) {
            if (i == 0)
               p = strtok(attachment_all, ",");
            else
          p = strtok(NULL, ",");

        if (p != NULL)
          strcpy(attachment[i], p);
        else
          break;
      }
    }
  }

  if (locked_by)
    el_decode(message, "Locked by: ", locked_by, 80);

  if (draft)
    el_decode(message, "Draft: ", draft, 80);

  p = strstr(message, "========================================\n");

  /* check for \n -> \r conversion (e.g. zipping/unzipping) */
  if (p == NULL)
    p = strstr(message, "========================================\r");

  if (text) {
    if (p != NULL) {
      p += 41;
      if ((int) strlen(p) >= *textsize) {
        strlcpy(text, p, *textsize);
        show_error("Entry too long to display. Please increase TEXT_SIZE and
            recompile elogd.");
        xfree(message);
        return EL_FILE_ERROR;
      } else {
        strlcpy(text, p, *textsize);

        /* strip CR at end */
        if (text[strlen(text) - 1] == '\n') {
          text[strlen(text) - 1] = 0;
          if (text[strlen(text) - 1] == '\r')
            text[strlen(text) - 1] = 0;
        }

        *textsize = strlen(text);
      }
    } else {
      text[0] = 0;
      *textsize = 0;
    }
  }

  xfree(message);
  return EL_SUCCESS;
}
```

Appendix B Implementing ELOG multi-instance support

Use el_build_index(&lb_list[logbook index], TRUE) to trigger indexing of logbook. Do this either periodically or try to use service similar to inotify but for nfs to trigger on file change.

Adding el_build_index(lbs, TRUE) to the start of show_elog_list function seems to handle synch without making logbooks noticeably slower from a user perspective.

```
// EXAMPLE
//Updates index of slave instances
if (getcfg(lbs->name, "Search slave", str, sizeof(str))){
  if(atoi(str)==1){
    if (get_verbose() >= VERBOSE_INFO)
    eprintf("Building %s index \n", lbs->name);

    el_build_index(lbs, TRUE);
  }
}
```

This gives lazy re-indexing which is faster but will not work if several instances are writing to the same logbook. This is not a problem if we boot all instances but one in read-only mode and set ′Search slave = 1′ in the global scope of the .cfg file used.