APPLICATION LOGGING FRAMEWORK FOR CONFD

tail-f Tail-f is now part of Cisco. cisco

Table of Contents

1. Abstract 4

2. Background 5

3. traceh.h 6

3.1. Installation 6

3.2. Usage 6

3.3. Logging levels 7

3.4. Macros 7

3.5. Log level controlled code block 8

3.6. Log file record 9

3.7. C interface to traceh.h 10

3.8. Performance 11

3.8.1 Performance of complied logging 11

3.8.2 Performance of runtime controlled logging 11

3.8.3 Timestamp format 11

3.9. Dependencies 11

4. Integration with a ConfD application 12

4.1. Data model 12

4.1.1 Configuration 12

4.1.2 State 13

4.2. CDB Subscriber 14

4.3. Validation 14

4.4. Data provider 15

4.5. CLI example 16

5. Diff (Meld) pattern 17

6. Conclusion 17

7. Source code 17

1. Abstract

ConfD provides applications developers many tools to use when developing ConfD client applications. Among the most important tools that an application developer needs are those which provide application debugging information. When working with ConfD, the application developer can generate many different log files, e.g., **devel.log**, **audit.log**, **netconf.log**, etc. These log files provide a great deal of information about processing which is occurring within the ConfD server daemon process. The developer can also use the API trace facility to examine the interaction between their application and the ConfD server daemon process. However, ConfD does not provide application logging services.

Application logging is usually implemented by its own code, e.g. by using a **printf()** like function. When a new application is developed, this approach is often repeated without reuse or with simple copy/paste reuse.

This application note presents a simple application logging framework called **traceh.h** which can be reused in ConfD C and C++ applications. Additionally, an example is presented of using ConfD along with a YANG data model to dynamically control and examine the state of the application logging framework at runtime.

2. Background

The logging framework has following features:

- supports standard logging levels (**FATAL**, **WARN**, **INFO**, **DEBUG** and **TRACE**)
- lightweight (small header file **traceh.h**)
- simple installation and usage
- logging level control
  - compile time control
  - runtime control
- log record contains important information such as:
  - timestamp
  - thread id
  - level
  - source file name with line number
  - message
- output to:
  - stdout/stderr
  - log file
  - syslog
- zero or minimal processing cost
- optionally - runtime levels can be easily configured through a YANG data model and application using ConfD

3. traceh.h

The **traceh.h** application logging framework for C/C++ is lightweight and consists of a single header file. The framework uses a set of predefined macros and functions.

3.1 Installation

Installation is simple. Copy the traceh.h source code file into the appropriate source code location of your project so it can be used with the #include preprocessor directive.

There is no need to add any object files to the build system (e.g. Makefile) or to link any additional libraries.

3.2 Usage

The header file should be included (**#include "traceh.h"**) in a C/C++ source code file. traceh.h contains definitions of some functions to be compiled. If traceh.h is included in multiple C/C++ source code files of an application, then one, **and only one**, of those source code files must precede the #include statement with **#define _TRACE_ DECLARE**.

Example:

```
#define _TRACE_DECLARE
#include "traceh.h"
```

The traceh.h application logging framework can be used without any initialization or setup.

The logging levels are controlled at compile time by setting the **T_LOG_<level>** preprocessor definition (e.g. **-DT_LOG_DEBUG**) or at runtime by calling the function **set_trace_log_level()**.

NOTE: The runtime logging level cannot be set to be more detailed than the compiled logging level.

For a simple application consisting of only one source file, it is possible to change the compiled logging level simply by defining the logging level before the #include "**traceh.h**" statement. While this approach does work, it is not recommended.

Example:

```
#define T_LOG_DEBUG
#define _TRACE_DECLARE
#include "traceh.h"
```

3.3 Logging levels

Supported logging levels correspond to the levels found in other common logging frameworks (e.g. SLF4J, Log4j, JCL, etc.).

The following are the supported logging levels and recommendation for their usage:

- **FATAL** - **severe errors**, runtime errors and unexpected conditions that cause premature termination or application malfunction
- **WARN** - use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong"
- **INFO** - interesting **runtime events** (startup/shutdown)
- **DEBUG** - **detailed information** on the flow through the system
- **TRACE** - **very detailed information**, should be run only when troubleshooting or verifying implementation flow during development

NOTE: FATAL should be used instead of the sometimes used **ERROR** logging level which is not supported.

3.4 Macros

Available macros for logging correspond to the logging levels and can be used with a printf-like format string. The macros will automatically append a new line character.

Example:

```
INFO("Processing starts");
DEBUG("Resurce id=%i", res_id);
TRACE("filename=%s", filename);
WARN("Expecting value <100, val=%i", val);
FATAL("NULL pointer, p_val=%p", p_val);
```

INFO, **DEBUG** and **TRACE** levels also have a **<level>_ENTER and <level>_EXIT** variant, which can be used to mark entry and exit points; usually to and from a function call. When used, it is recommended to follow the rule that a function has exactly one exit point.

Example:

```
int count(int times) {
    TRACE_ENTER("times=%i", times);
    int ret = 0;

    ...

    TRACE_EXIT("ret=%i", ret);
    return ret;
}
```

The log record will contain ==> (entry) and <== (exit) marks.

Example:

```
1487234002.158580 [7fd098710700] TRACE file.c:10: ==> count times=20
1487234002.158619 [7fd098710700] TRACE file.c:20: <== count ret=1
```

In addition, for the **TRACE** level, there exists the **TRACE_BEGIN** and **TRACE_END** variants.

Example:

```
TRACE_BEGIN("Reading started");
...

TRACE_END("Reading ended, bytes=%i", bytes);
```

The log record will contain **-->** (entry) and **<--** (exit) marks.

```
1487234002.158580 [7fd098710700] TRACE /file.c:10: --> Reading started
1487234002.158619 [7fd098710700] TRACE /file.c:20: <-- Reading ended,
bytes=4569
```

It is important to mention that **TRACE**, **DEBUG** and **INFO** messages are printed on **stdout** and the **WARN** and **FATAL** messages on **stderr**.

3.5 Log level controlled code block

Sometimes it is necessary to compile some code specific to the enabled log level.

To check what compiled log level has been set during compilation, use the defines **T_LOG_TRACE**, **T_LOG_DEBUG**, **T_LOG_INFO**, **T_LOG_WARN** and **T_LOG_FATAL**.

Example:

```
#if defined(T_LOG_DEBUG) || defined(T_LOG_TRACE)
    char buffer[256];
    confd_pp_kpath(buffer, buff_len, path);
    buffer[buff_len] = '\0';
    DEBUG("%s keypath == %s", str, buffer);
#endif
```

To check the log level at runtime, use the **get_trace_log_level()** function.

Example:

```
if (get_trace_log_level() >= LOG_DEBUG) {
    char buffer[256];
    confd_pp_kpath(buffer, buff_len, path);
    buffer[buff_len] = '\0';
    DEBUG("%s keypath == %s", str, buffer);
}
```

NOTE: Notice that the example uses "**>=**". It is important to include a check for all levels with higher verbosity (details) and not just for the level that we are interested in.

3.6 Log file record

The log record contains:

- UTC timestamp - formatted **(03-Mar-2017::10:59:30.008 431us)** or simple **(seconds and microseconds - 1488535170.008458)**
- thread id (pthread)
- log level - corresponds to the level string , e.g. **TRACE**, **INFO** (except for **FATAL** which is reported as **CRIT**)
- full path of sourcefile
- line in the sourcefile
- optional entry, exit marks **(==>, <==, -->, <--)**
- message - corresponds to the printed message

Example:

```
TRACE("Create socket");
```

Produces following log entry line:

```
03-Mar-2017::10:59:30.008 431us [7f16cf262700] TRACE status.c:171: Create
socket
```

or (when simple time format is used):

```
1488535170.008458 [7f16cf262700] TRACE status.c:171: Create socket
```

3.7 C interface to traceh.h

In addition to the logging macros, the **traceh.h** application logging framework also provides the following interface functions:

```
extern void init_syslog_trace(const char *ident, int option, int facility);
extern void init_trace_streams(FILE* strout, FILE* strerr);
extern void set_trace_log_level(int level);
extern void set_trace_stream_level(int level);
extern void set_trace_syslog_level(int level);
extern void set_trace_time_format(enum time_format format);
extern int get_trace_log_level();
extern int get_trace_stream_level();
extern int get_trace_syslog_level();
extern int get_trace_log_compiled_level();
extern enum time_format get_trace_time_format();
extern int is_syslog_initialized();
extern int are_streams_initialized();
extern const char *get_trace_level_name(const int level);
```

These functions can be used to initialize logging of messages to the syslog facility **(init_syslog_trace())** or to file streams **(init_trace_streams())**. It is possible to query if syslog or streams have been initialized using the functions **is_syslog_initialized()** and **are_streams_initialized()**.

There are also functions which can be used to set runtime logging levels for console, syslog or streams. Value LOG_OFF can be used to turn off logging completely. These functions are **set_trace_log_level()**, **set_trace_stream_level()**, and **set_trace_stream_level()**.

The functions **get_trace_log_level()**, **get_trace_stream_level()**, **get_trace_syslog_level()** and **get_trace_log_compiled_level()** query the individual log levels and check what the compiled log level is (i.e. the base level for all types of logging - i.e. syslog, streams or console).

The function **get_trace_level_name()** converts the numeric log level to a string value.

The function **set_trace_time_format()** can change time format from formatted (default - e.g. 03-Mar-2017::10:59:30.008 431us) to **simple** (1488535170.008458) and back. The corresponding **get_trace_time_format()** function returns the current time format.

The log levels are defined via #define with integer values:

```
LOG_OFF // -1
 LOG_EMERG // 0
 LOG_ALERT // 1
 LOG_CRIT // 2  -> FATAL
 LOG_ERR // 3
 LOG_WARNING // 4 -> WARN
 LOG_NOTICE // 5
 LOG_INFO // 6   -> INFO
 LOG_DEBUG // 7  -> DEBUG
 LOG_TRACE // 10 -> TRACE
```

NOTE: The defined log levels correspond to the system syslog levels and not all have a macro equivalent. The framework uses only the following levels: **LOG_OFF**, **LOG_CRIT** (corresponds to **FATAL**), **LOG_WARNING** (corresponds to **WARN**), **LOG_INFO, LOG_DEBUG**, and **LOG_TRACE**. The use of other log levels (**LOG_EMERG, LOG_ALERT**, **LOG_ERR**, and **LOG_NOTICE**) should be avoided.

3.8 Performance

3.8.1 Performance of compiled logging

The compiled logging, as determined by either **-DT_LOG_<level>** or **#define T_LOG_<level>**, has no performance penalty when the corresponding logging level is not active. The logging lines simply disappear from resulting binary because they are a conditional compilation.

3.8.2 Performance of runtime controlled logging

The runtime time logging, as set, for example, by the **set_trace_log_level()** function) has negligible performance penalty when the corresponding logging level is not active. The performance penalty is equal to the comparison of an integer value.

Any processing of the text formatting inside the logging macros does not influence performance, when the corresponding logging level is not active. Thus, any time consuming calls can be used in the formatter.

Example:

```
TRACE("Result of calculation %i", time_consuming_function());
```

When **TRACE** level is not active, **time_consuming_function()** will not be called.

3.8.3 Timestamp format

By default, **formatted** timestamp is used in the log records. For extensive logging, there may be some time penalty associated with this formatting (approximately 1-2 microseconds). If this is the case, then the **simple** timestamp format can be used (see **set_trace_time_format()**).

3.9 Dependencies

The application logging framework depends on a few standard unix/linux header files and posix threads:

```
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>
#include <stdarg.h>
#include <syslog.h>
```

These files are automatically included by **traceh.h** header file.

NOTE: While the application logging framework does include **pthread.h**, the framework does not create any threads. The **pthread_mutex_*()** functions are used along with **pthread_self()**.

4. Integration with a ConfD application

An example of a possible way to integrate the **traceh.h** application logging framework with ConfD is available.

This example integration controls the configuration of the logging levels and timestamp format through the ConfD northbound interfaces, performs validation, and provides the user with information about current logging levels and timestamp format as operational data.

4.1 Data Model

4.1.1 Configuration

The configuration portion of the example YANG data model:

```
container logging {
    description "Configuration parameters for traceh.h framework.";

    leaf console-level {
      type log-level-type;
      description "Log level configuration for console logging.";
      tailf:validate logging_config_val {
        tailf:dependency .;
      }
    }

    leaf stream-level {
      type log-level-type;
      description "Log level configuration for stream logging.";
      tailf:validate logging_config_val {
        tailf:dependency .;
      }
    }

    leaf syslog-level {
      type log-level-type;
      description "Log level configuration for syslog logging.";
      tailf:validate logging_config_val {
        tailf:dependency .;
      }
    }

    leaf time_format {
      type log-time-format;
      description "Time format configuration for logging.";
    }
  }
```

Each of the logging types (**console**, **syslog**, and **streams**) has a corresponding leaf. The validation point is reused in multiple locations.

4.1.2 State

The state (operational data) portion of the example YANG data model:

```
container logging-state {
    config false;
    description "State data of the traceh.h framework.";
    tailf:callpoint logging_state_dp;

    leaf console-level {
      type log-level-type;
      description "Log level for console logging.";
    }

    leaf stream-level {
      type log-level-type;
      description "Log level for stream logging.";
    }

    leaf syslog-level {
      type log-level-type;
      description "Log level for syslog logging.";
    }

    leaf compiled-level {
      type log-level-type;
      description "Log level for compiled logging.";
    }

    leaf time_format {
      type log-time-format;
      description "Time format for logging.";
    }

    leaf syslog_initialized {
      type boolean;
      description "Is syslog logging initialized?";
    }

    leaf streams_initialized {
      type boolean;
      description "Is streams logging initialized?";
    }
  }
```

Each of the logging types (**console**, **syslog**, and **streams**) has a corresponding leaf. In addition, the YANG data model provides information about the compiled log level and **syslog** and **streams** initialization. The data provider is registered on the top level container.

4.2 CDB Subscriber

The CDB subscriber listens for changes in the configuration and sets the changes in the **traceh.h** application logging framework.

Example implementation:

```
if (cdb_get_modifications(subsock, logging_spoint, 0, &values,
                          &nvalues,
                          NULL) == CONFD_OK) {
    INFO("Logging subscription triggered");
    for (i = 0; i < nvalues; i++) {
        if (values[i].tag.tag == logging_console_level) {
            TRACE("Processing console level");
            val = CONFD_GET_ENUM_VALUE(&values[i].v);
            TRACE("val=%i", val);
            set_trace_log_level(val);
        }
        if (values[i].tag.tag == logging_syslog_level) {
            TRACE("Processing syslog level");
            val = CONFD_GET_ENUM_VALUE(&values[i].v);
            TRACE("val=%i", val);
            set_trace_syslog_level(val);
        }
        if (values[i].tag.tag == logging_stream_level) {
            TRACE("Processing stream level");
            val = CONFD_GET_ENUM_VALUE(&values[i].v);
            TRACE("val=%i", val);
            set_trace_stream_level(val);
        }
        if (values[i].tag.tag == logging_time_format) {
            TRACE("Processing time format");
            val = CONFD_GET_ENUM_VALUE(&values[i].v);
            TRACE("val=%i", val);
            set_trace_time_format(val);
        }
        confd_free_value(CONFD_GET_TAG_VALUE(&values[i]));
    }
    free(values);
}
```

4.3 Validation

The validation checks if the runtime logging level being set is not more detailed than the compiled logging level.

Example of implementation:

```
int new_level = CONFD_GET_ENUM_VALUE(newval);
TRACE("validation new_level=%i", new_level);

if (new_level > get_trace_log_compiled_level()) {
    WARN("Validation issue level %i compiled level %i", new_level,
            get_trace_log_compiled_level());
    char ebuf[BUFSIZ];
    sprintf(ebuf, "Level %s cannot be set as it is more detailed than"
            " compiled level %s!", get_trace_level_name(new_level),
            get_trace_level_name(get_trace_log_compiled_level()));
    confd_trans_seterr(tctx, "%s", ebuf);
    ret = CONFD_ERR;
}
```

4.4 Data provider

The data provider gets the current operational data values from the **traceh.h** application logging framework and returns them.

Example of implementation:

```
static int get_elem(struct confd_trans_ctx *tctx, confd_hkeypath_t *keypath)
{
    TRACE_ENTER("");
    int ret = CONFD_OK;
    confd_value_t v;
    int val;

    switch (CONFD_GET_XMLTAG(&(keypath->v[0][0]))) {
    case logging_console_level:
        TRACE("console level");
        val = get_trace_log_level();
        CONFD_SET_ENUM_VALUE(&v, val);
        break;
    case logging_stream_level:
        TRACE("stream level");
        val = get_trace_stream_level();
        CONFD_SET_ENUM_VALUE(&v, val);
        break;
    case logging_syslog_level:
        TRACE("syslog level");
        val = get_trace_syslog_level();
        CONFD_SET_ENUM_VALUE(&v, val);
        break;
    case logging_compiled_level:
        TRACE("compile level");
        val = get_trace_log_compiled_level();
        CONFD_SET_ENUM_VALUE(&v, val);
        break;
    case logging_syslog_initialized:
        TRACE("syslog initialized");
        val = is_syslog_initialized();
        CONFD_SET_BOOL(&v, val);
        break;
    case logging_streams_initialized:
        TRACE("streams initialized");
        val = are_streams_initialized();
        CONFD_SET_BOOL(&v, val);
        break;
    case logging_time_format:
        TRACE("time format");
        val = get_trace_time_format();
        CONFD_SET_ENUM_VALUE(&v, val);
        break;
    }
```

```
    TRACE("get_elem value found val=%i", val);
    confd_data_reply_value(tctx, &v);

    TRACE_EXIT("ret %i", ret);
    return ret;
}

static int get_next(struct confd_trans_ctx *tctx, confd_hkeypath_t *keypath,
        long next)
{
    TRACE_ENTER("");
    int ret = CONFD_ERR;
    FATAL("As data model has no list, the 'get_next' should not be called!");
    TRACE_EXIT("ret %i", ret);
    return ret;
}
```

4.5 CLI example

Example of possible usage in the C-style CLI when the application is compiled with the **INFO** log level:

```
    # show logging-state
logging-state console-level info
logging-state stream-level info
logging-state syslog-level info
logging-state compiled-level info
logging-state syslog_initialized false
logging-state streams_initialized false
# config
Entering configuration mode terminal
(config)# logging console-level warn
# commit
Commit complete.
# do show logging-state
logging-state console-level warn
logging-state stream-level info
logging-state syslog-level info
logging-state compiled-level info
logging-state syslog_initialized false
logging-state streams_initialized false
# logging console-level trace
(config)# commit
Aborted: 'logging console-level': Level TRACE cannot be set as it is more
          detailed than compiled level INFO!
(config)#
```

5. Diff (Meld) pattern

The following regexp can be used to ignore the timestamp (**in simple format**) and thread id when comparing two log files (e.g. same run sequence at different times).

```
^[0-9]+\.[0-9]+ \[.+\]
```

This pattern can be used in the Meld diff tool **(Preferences –> Text Filters)**.

6. Conclusion

The **traceh.h** application logging framework provides an easy way to enable logging in ConfD C and C++ applications. It is minimalistic and simple to use. It provides a basic tool which is useful while developing and troubleshooting applications. Because there are no performance and binary size penalties for compiled logging levels the application logging framework can be used in real time code and embedded systems.

7. Source code

The source code for the discussed example ConfD integration with the **traceh.h** application logging framework which includes the **traceh.h** source code is available in the ConfD distribution's examples set as of ConfD v6.5. Prior to then, please, contact your Tail-f Solutions Architect in order to obtain a copy.

www.tail-f.com
info@tail-f.com

Corporate Headquarters

Sveavagen 25
111 34 Stockholm
Sweden
+46 8 21 37 40