НТЦ Метротек

«Научно-технический центр Метротек»

Лабораторная работа №1

Тема: Сбор статистики по сетевому трафику

Стажёр: Бронников Егор Игоревич

<bronnikov.40@mail.ru>
<t.me/endygamedev>

Санкт-Петербург

2022

Содержание

Постановка задачи

Цель задачи: создать набор программного обеспечения, который мог бы собирать и отображать статистику по трафику на заданном сетевом интерфейсе.

Требования:
1. ПО должно работать на ПК под управлением Debian GNU\Linux (версии 10 и новее).
2. Для реализации использовать язык программирования C.
3. Сборка должна осуществляться GNU Toolchain.
4. Дистрибуция должна осуществляться при помощи deb-пакета.
5. Сбор статистики должен вестись только по входящим UDP пакетам.
6. Должна быть реализована возможность указывать конкретные параметры учитываемых в статистике пакетов:
    1. IP-адрес источника
    2. IP-адрес назначения
    3. Порт источника
    4. Порт назначения
7. В статистике должно присутствовать количество принятых пакетов и суммарное количество байт в этих пакетах.

ПО организовать в виде двух отдельных утилит: первая читает данные с сетевого интерфейса и собирает статистику по пакетам, вторая при запуске получает собранную статистику у первой утилиты и выводит её на экран.

Утилита для сбора статистики. Нужно реализовать 2 варианта данной утилиты:
1. Два потока (pthread): первый читает пакеты при помощи Raw Socket (OSI L2) с интерфейса, проверяет параметры пакета и для

подходящих по заданным параметрам, передаёт статистику во второй поток. Второй суммирует статистику и отдаёт её по запросу извне.
2. Два потока (pthread): первый читает пакеты при помощи Raw Socket (OSI L2) с интерфейса, проверяет параметры пакета и для подходящих по заданным параметрам, суммирует статистику. Второй отдаёт её по запросу извне.

Самостоятельно провести профилирование обоих вариантов, оценить какой вариант эффективнее, с каким вариантом можно обеспечить большую пропускную способность.
В обоих вариантах: передача данных между потоками осуществляется любым способом, на усмотрение разработчика.
Утилита для вывода статистики на экран: запрашивает статистику у первой утилиты через ubus или через POSIX Message Queues. Рекомендация: попробовать реализовать оба варианта.

Программное обеспечение должно сопровождаться документацией, содержащей следующие разделы:
1. Описание - общая информация, что и как делает ПО.
2. Сборка - инструкции по сборке ПО из исходников: что установить в систему, какой командой запустить сборку, что должно получиться в итоге.
3. Запуск - как запустить ПО, как подать трафик на интерфейс, чтобы убедиться в корректности работы, что пользователь программ должен увидеть на экране.
4. Результаты профилирования двух реализованных вариантов утилиты для сбора статистики.
5. Авторство и лицензия - указать имя и электронную почту автора, указать лицензию.
Результат работы: архив git репозитория, содержащего исходники ПО и сопроводительную документацию.

Вспомогательная информация:

- The Linux Programming Interface: https://man7.org/tlpi/
- GCC: https://gcc.gnu.org
- pthread: https://man7.org/linux/man-pages/man7/pthreads.7.html
- Message Queues: https://man7.org/linux/man-pages/man7/mq_overview.7.html
- Raw Sockets: https://man7.org/linux/man-pages/man7/raw.7.html
- ubus: https://openwrt.org/docs/techref/ubus
- Сборка deb-пакета: https://www.debian.org/doc/devel-manuals#debmake-doc
- Профилирование: https://www.brendangregg.com/
- Git: https://git-scm.com/book/en/v2

1. Описание

2. Сборка

3. Запуск

4. Профилирование

5. Авторство и лицензия

Приложение А

Coding Style

C

В сторонних проектах с собственным описанным стилем оформления кода следует придерживаться правил этого стиля:

- U-Boot: https://www.denx.de/wiki/U-Boot/CodingStyle
- Linux: https://www.kernel.org/doc/html/v4.10/process/coding-style.html

В сторонних проектах без описанного стиля следует оформлять наш код по аналогии с остальными исходниками проекта.

В наших собственных проектах за основу взят стиль, принятый в linux с некоторыми изменениями.

Отступы

В качестве отступа используется один таб шириной в четыре пробела.

```
int main(void)
{
    int a = 0;
    return a;
}
```

Лишние табы и пробелы в конце строк следует удалять.

Длина строк

Следует избегать превышения ограничения в 80 символов на строку. Если выражение не помещается в 80 символов, его следует разделить

на части. При этом желательно, чтобы каждый аргумент функции находился в отдельной строке. Пример:

```
return_code = my_structure->callback_with_arguments(argument_number_1,
                                                    argument_number_2);
```

Строки, выводимые программой на экран, не следует сокращать при превышении ими ограничения.

Скобки

В условиях, циклах и объявлениях структур открывающая фигурная скобка не переносится на следующую строку. Закрывающая переносится:

```
while (a > b) {
    a--;
    b++;
    if (a == b) {
        do_something();
    }
}
```

Закрывающая фигурная скобка располагается на том же уровне отступов, что и начало всей конструкции.

В объявлениях функций открывающая фигурная скобка переносится на следующую строку:

```
int main(int argc, char **argv)
{
    return 0;
}
```

В условиях и циклах перед открывающей круглой скобкой ставится пробел. После закрывающей скобки ставится пробел:

```
if (a == 0) {
    return a;
}
```

Фигурные скобки должны присутствовать даже если в блоке всего одно выражение.

Пробелы

Бинарные и тернарные операторы окружаются пробелами:

```
= + - < > * / % | & ^ <= >= == != ? :
```

После унарных операторов не ставится пробел:

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

Пробел не ставится после префиксных инкремента и декремента:

```
++i
--i
```

и перед постфиксными:

```
i++
i--
```

Пробелы не ставятся вокруг операторов структур:

```
my_struct.element
my_struct->element
```

При объявлении указателя оператор * идет перед именем указателя, а не после его типа:

```
void *buffer
```

Имена и объявления

В именовании функций, переменных и типов используется *snake_ case*. Примеры:

```
void my_function(int my_argument);
struct my_struct *s;
```

В именах желательно стараться избегать сокращений, из-за которых становится не ясно назначение сущности. Например, вместо **dctl()** использовать более полное имя **device_control()**.

Объявлять переменные в функции желательно рядом с выражениями, их использующими, если функция достаточно длинная (от 10 строк), либо в начале функции, если короткая (до 10 строк). Например, переменную-счётчик для цикла for следует объявлять так:

```
for (int i = 0; i < 3; ++i) {
    actions();
}
```

Также нужно следить за количеством переменных в одной функции. Если их число превышает пять, то следует задуматься о декомпозиции функции.

Функции

Функции должны быть как можно короче и выполнять как можно меньше действий. Длинные функции следует разбивать на подфункции, действия внутри которых связаны по смыслу. Пример:

```
int collect_data_and_calculate_result(struct program_context *c)
{
    void *data = collect_data(c);

    return calculate_result(data);
}

int init_and_run_program(struct arguments *a){
    struct program_context *c = init_context(a);

    return collect_data_and_calculate_result(c);
```

```
}
```

Структуры перечисления

Их объявления должны выглядеть следующим образом.
Структуры только с именем:

```
struct my_struct_name {
    int i;
    int j;
    void *buffer;
};
```

Структуры с typedef должны иметь суффикс _t в имени типа:

```
typedef struct my_struct {
    int i;
    int j;
    void *buffer;
} my_struct_t;
```

Перечисление объявляются аналогичным образом. Все варианты перечисления должны именоваться в верхнем регистре:

```
typedef enum {
    ADDRESS_7BIT,
    ADDRESS_10BIT,
} addressing_mode_t;
```

Макросы

Желательно избегать написания макросов. Особенно вложенных, так как это приводит к сложностям в отладке.

Макросы именуются с использованием только верхнего регистра. Макрос и отдельно его входные параметры должны быть окружены круглыми скобками:

```
#define MIN(x, y)     ((x) < (y) ? (x) : (y))
```

Макросы с несколькими выражениями должны быть заключены в блок `do {} while(0)`:

```
#define DO_ACTION(a, b)                          \
                do {                             \
                    if (do_first(a) >= 0) {      \
                    do_next(b);                  \
                    }                            \
                } while(0)                       \
```

Использование goto

`goto` можно использовать для обработки ошибок. Переходя должны осуществляться только в пределах одной функции. Пример:

```
int configure_device(void)
{
    int ret = 0;
    struct my_device *d = malloc(sizeof(my_device));
    if (!d) {
        return -ENOMEM;
    }

    ret = init_stage_first(d);
    if (ret != 0) {
        goto err_free;
    }

    ret = init_stage_last(d);
    if (ret != 0) {
        goto err_free;
    }

    return 0;

err_deinit:
```

```
    deinit_stage_first(d);
err_free:
    free(d);
    return ret;
}
```

Комментарии

Комментарии следует использовать только в качестве документации и для пояснения каких-то не очевидных специфичных случаев.

Функции

Документировать следует функции, поведение, входные и выходные параметры которые не очевидны. Пример:

```
/*
 * Configures an I2C bus.
 *
 * addr_lenght should be ADDRESS_7BIT or ADDRESS_10BIT.
 * speed should be SPEED_100MHZ or SPEED_400MHZ.*
 * Returns 0 on success, -1 otherwise.
 */
int configure_i2c_bus(int bus_number, int addr_length, int speed)
```

Не следует документировать очевидные функции. Например, в достаточно очевидны назначение и принцип работы функции. Пример:

```
int sum_int(int a, int b)
{
    return a + b;
}
```

Магические числа

Если назначение числового значения не очевидно из имени переменной или макроса, которым оно присваивается, то это значение следу-

ет снабдить комментарием. Пример:

```
#define DEVICE_CONTROL_REGISTER_ADDR 0x4
/* Set required device mode on startup. See device datasheet page
   100, table 20. */
#define DONTROL_REGISTER_DATA 0x4f7a
```

Заголовочные файлы

Должны обязательно содержать защиту от повторного включения:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
All the contents of the header file here
#endif
```

либо

```
#pragma once
Content
```

Содержимое может включать:

- дополнительные `include`
- объявление макросов
- объявление констант
- прототипы функций

но реализации функций должны содержаться только в файлах `.c`

Python

См. https://peps.python.org/pep-0008/