Лабораторная работа № 6
по курсу "Операционные системы":

Выполнил студент группы 08-208 МАИ *Куликов Алексей.*

Цель работы

Целью является приобретение практических навыков в:

1. Управлении серверами сообщений
2. Применение отложенных вычислений
3. Интеграция программных систем друг с другом

Задание

Реализовать клиент-серверную систему по асинхронной обработке запросов. Необходимо составить программы сервера и клиента. При запуске сервер и клиент должны быть настраиваемы, то есть должна быть возможность поднятия на одной ЭВМ нескольких серверов по обработке данных и нескольких клиентов, которые к ним относятся. Все общение между процессами сервера и клиентов должно осуществляться через сервер сообщений.

Серверное приложение – банк. Клиентское приложение клиент банка. Клиент может отправить какую-то денежную сумму в банк на хранение. Клиент также может запросить из банка произвольную сумму. Клиенты могут посылать суммы на счета других клиентов. Запросить собственный счет. При снятии должна производиться проверка на то, что у клиента достаточно денег для снятия денежных средств. Идентификатор клиента задается во время запуска клиентского приложения, как и адрес банка. Считать, что идентификаторы при запуске клиентов будут уникальными.

В качестве конкретного варианта задания предлагается создание клиент-серверной системы, удовлетворяющей всем вышеуказанным условиям, а так же следующим, зависящим от варианта задания(вариант 30):

- В качестве внутреннего хранилища сервера использовать бинарное дерево, где ключом является идентификатор клиента.
- Тип ключа клиента – строка.
- Дополнительная возможность сервера – возможность временной приостановки работы сервера без выключения. Сообщения серверу можно отправлять, но ответы сервер не отправляет до возобновления работы.

.

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

(*) Дополнительное задание. Необходимо запустить клиентские приложения и сервера на разных виртуальных машинах или в разных Docker контейнерах.

Описание программы

Система, по сути, две разные программы, которые могут взаимодействовать друг с другом благодаря библиотеке ZeroMQ. Одна программа – это сервер, другая – клиент. Экземпляры программ-клиентов могут взаимодействовать с программами-серверами посредством очереди сообщений, предоставляемой библиотекой ZeroMQ.

Файлы `request.h`, `response.h` содержат структуру запроса и ответа соответственно, а так же определения для команд.

Файл `server.c` содержит, очевидно, логику работы серверного приложения.

Сервер работает следующим образом. Сервер загружает импровизированную базу данных (в моем случае бинарное дерево) из файла, если ему предоставлен путь. Далее создаеются контекст и сокет при помощи `zmq_ctx_new()` и `zmq_socket()`. Сокет связывается с конкретным портом(так же передается в аргументе) вызовом `zmq_bid()` и ждет последующих подключений. Далее начинается основной цикл работы сервера. Сервер берет на обработку запрос из очереди сообщений при помощи `zmq_recv()`. Потом он производит необходимые действия (клиент может положить на счет, снять, перевести другому клиенту деньги или запросить баланс) и формирует ответ на запрос (успех/-неуспех). Далее ответ отправляется клиенту, который делал этот запрос, при помощи `zmq_send()`.

В программе-сервере так же определены пользовательские обработчики системных сигналов `SIGINT` и `SIGTSTP` для остановки сервера и временной приостановки соответственно.

В конце работы сервера, когда произошло прерывание по сигналу `SIGINT`, сервер сохраняет данные в базу данных и освобождает ресурсы, закрывая сокет и уничтожая контекст (вызовы `zmq_close()` и `zmq_ctx_destroy()`).

Файл `client.c` содержит логику работы клиентского приложения.

В клиентском приложении происходит следующее. Так же создается контекст и сокет. Далее из переданного в качестве аргумента порта формируется адрес сервера, и сокет соединятся с сервером при помощи `zmq_connect()`. Начинается основной цикл выполнения. Приложение ждет пользовательских указаний, после их получения формируется запрос. Далее запрос отправляется серверу, и пользователь ждет ответа на запрос. Когда ответ получен, приложение печатает результат выполнения банковской операции. Потом все повторяется.

Когда пользователь изволит закончить работу вводом `exit`, программа выходит из бесконечного цикла, освобождает ресурсы и завершается.

Листинг

server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <zmq.h>
#include <assert.h>
#include <signal.h>

#include "request.h"
#include "response.h"
#include "binary_tree.h"

#define RUN 0
#define PAUSE 1
#define CLOSE 2

int control = RUN;

void sigtstp_handler(int sig);
void sigint_handler(int sig);

int main(int argc, char **argv){

    tree *accounts = NULL;
    if(argc < 2){
        printf("usage: server <port> [account_data]\n");
        exit(EXIT_SUCCESS);
    }
    if(argc == 3){
        printf("loading data from %s ...\n", argv[2]);
        FILE *inp = fopen(argv[2], "r");
        assert(inp != NULL);
        accounts = deserialize_tree(inp);
        fclose(inp);
    }
    signal(SIGTSTP, sigtstp_handler);
    signal(SIGINT, sigint_handler);

    print_tree(accounts, 0);

    char adress[30];
    sprintf(adress, "tcp://*:%s", argv[1]);

    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_REP);
    int status = zmq_bind(socket, adress);
    assert(status == 0);

    while (control != CLOSE){
        while (control == PAUSE){
            printf("\nlunch break!");
            sleep(1);
        }

        request req;
        int status = zmq_recv(socket, &req, sizeof(request), 0);
        if(status){
            printf("status: %d\n", status);
            if (status == EAGAIN || status == EINTR){
                continue;
            }
            else{
```

```
            fprintf(stderr, "server::recv: %d\n", zmq_errno());
            exit(EXIT_FAILURE);
        }
    }

    if(control == PAUSE || control == CLOSE)
        continue;

    tree *found = find_tree(accounts, req.client1);
    if (found == NULL){
        printf("new client: %s\n", req.client1);
        account temp;
        strcpy(temp.owner, req.client1);
        temp.balance = 0;
        accounts = insert_tree(accounts, &temp);
        found = find_tree(accounts, req.client1);
    }

    response res;
    switch (req.oper){
    case DEPOSIT:
        printf("%s deposited %d\n", req.client1, req.value);
        found->acc.balance += req.value;
        res.stat = OK;
        res.value = found->acc.balance;
        break;
    case WITHDRAW:
        printf("%s withdrawed %d\n", req.client1, req.value);
        int *balance = &found->acc.balance;
        if (*balance < req.value){
            res.stat = NOT_OK;
        }
        else{
            *balance -= req.value;
            res.stat = OK;
        }
        res.value = *balance;
        break;
    case TRANSFER:
        printf("%s transfered %d to %s\n", req.client1, req.value, req.client2);
        tree *targ = find_tree(accounts, req.client2);
        if (targ == NULL)
            res.stat = NOT_OK;
        else{
            int *balance = &found->acc.balance;
            if (*balance < req.value)
                res.stat = NOT_OK;
            else{
                *balance -= req.value;
                targ->acc.balance += req.value;
                res.stat = OK;
            }
            res.value = *balance;
        }
        break;
    case BALANCE:
        printf("%s requested balance\n", req.client1);
        res.stat = OK;
        res.value = found->acc.balance;
        break;
    default:
        res.stat = NOT_OK;
        res.value = found->acc.balance;
        break;
    }
```

```
        if(control == RUN)
            sleep(10);

        status = zmq_send(socket, &res, sizeof(response), 0);
        if (status == EAGAIN || status == EINTR) {}
        else{
            fprintf(stderr, "server::send: %d\n", zmq_errno());
            exit(EXIT_FAILURE);
        }
    }
    if (argc == 3){
        printf("\nsaving data to %s ...\n", argv[2]);
        FILE *op = fopen(argv[2], "w");
        assert(op != NULL);
        serialize_tree(accounts, op);
        fclose(op);
    }

    destroy_tree(accounts);
    zmq_close(socket);
    zmq_ctx_destroy(context);

    return 0;
}

void sigtstp_handler(int sig){
    if (control == RUN)
        control = PAUSE;
    else
        control = RUN;
}

void sigint_handler(int sig){
    control = CLOSE;
}
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <zmq.h>

#include "request.h"
#include "response.h"

void show_menu();

int main(int argc, char **argv){
    if(argc != 3){
        printf("usage: client <port> <client_name>\n");
        exit(EXIT_SUCCESS);
    }

    char adress[30];
    sprintf(adress, "tcp://localhost:%s", argv[1]);
    printf("adress: %s\n", adress);

    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_REQ);
    int status = zmq_connect(socket, adress);
    assert(status == 0);
```

```
char buffer[20];

printf("ask <help> if you don't know what to do\n");

while(1){
    scanf("%s", buffer);
    if(!strcmp("exit", buffer))
        break;
    if (!strcmp("help", buffer)){
        show_menu();
        continue;
    }

    int correct = 0;
    request req;

    strcpy(req.client1, argv[2]);
    if(!strcmp("deposit", buffer)){
        req.oper = DEPOSIT;
        scanf("%d", &req.value);
        correct = 1;
    }
    if(!strcmp("withdraw", buffer)){
        req.oper = WITHDRAW;
        scanf("%d", &req.value);
        correct = 1;
    }
    if(!strcmp("transfer", buffer)){
        req.oper = TRANSFER;
        scanf("%s", req.client2);
        scanf("%d", &req.value);
        correct = 1;
    }
    if(!strcmp("balance", buffer)){
        req.oper = BALANCE;
        req.value = 0;
        correct = 1;
    }
    if(correct){
        printf("Operation accepted. Waiting for completion.\n");
        zmq_send(socket, &req, sizeof(request), 0);

        response res;
        zmq_recv(socket, &res, sizeof(response), 0);

        switch (res.stat){
            case OK:
                printf("Status: OK. Operation completed.\n");
                break;
            case NOT_OK:
                printf("Status: NOT_OK. Operation denied.\n");
                break;
        }
        printf("Now your balance is: %d\n", res.value);
    }
    else{
        printf("Wrong request. Try again please\n");
    }
}

zmq_close(socket);
zmq_ctx_destroy(context);

return 0;
```

```
}

void show_menu(){
    printf("You can:\n");
    printf("deposit <sum>\n");
    printf("withdraw <sum>\n");
    printf("transfer <person> <sum>\n");
    printf("balance\n");
    printf("exit\n");
}
```

request.h

```
#ifndef REQUEST_H
#define REQUEST_H

enum{
    DEPOSIT,
    WITHDRAW,
    TRANSFER,
    BALANCE
};

typedef struct{
    char client1[20];
    char client2[20];
    int oper;
    int value;
} request;

#endif
```

response.h

```
#ifndef RESPONSE_H
#define RESPONSE_H

enum{
    OK,
    NOT_OK
};

typedef struct{
    int stat;
    int value;
} response;

#endif
```

binary_tree.h

```
#ifndef BINARY_TREE_H
#define BINARY_TREE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char owner[20];
    int balance;
} account;

typedef struct node_t{
    struct node_t *left, *right;
    account acc;
```

```
} tree;

tree* create_tree(account *acc){
    tree *temp = (tree*)malloc(sizeof(tree));
    temp->left = temp->right = NULL;
    temp->acc = *acc;
    return temp;
}

tree *insert_tree(tree *t, account *acc){
    if(t == NULL)
        return create_tree(acc);
    else if(strcmp(acc->owner, t->acc.owner) < 0)
        t->left = insert_tree(t->left, acc);
    else if(strcmp(acc->owner, t->acc.owner) > 0)
        t->right = insert_tree(t->right, acc);
    return t;
}

tree *find_tree(tree *t, char *name){
    while(t != NULL && strcmp(name, t->acc.owner) != 0){
        if(strcmp(name, t->acc.owner) < 0)
            t = t->left;
        else
            t = t->right;
    }
    return t;
}

tree* detach_min(tree* t){
    tree *par = t;
    tree *temp = t->right;
    while(temp->left != NULL){
        par = temp;
        temp = temp->left;
    }
    if(par == t)
        par->right = temp->right;
    else
        par->left = temp->right;
    return temp;
}

tree *erase_tree(tree *t, char *name){
    if(t == NULL)
        return NULL;
    else if (strcmp(name, t->acc.owner) < 0)
        t->left = erase_tree(t->left, name);
    else if (strcmp(name, t->acc.owner) > 0)
        t->right = erase_tree(t->right, name);
    else{
        tree *temp;
        if (t->left == NULL && t->right)
            temp = t->right;
        else if(t->right == NULL)
            temp = t->left;
        else{
            tree *m = detach_min(t);
            t->acc = m->acc;
            free(m);
            return t;
        }
        free(t);
        return temp;
    }
```

```
    return t;
}

void print_tree(tree *t, int tab){
    if(t == NULL)
        return;
    print_tree(t->left, tab + 2);
    for(int i = 0; i < tab; i++)
        printf("%c", ' ');
    printf("%s : %d \n", t->acc.owner, t->acc.balance);
    print_tree(t->right, tab + 2);
}

void destroy_tree(tree *t){
    if(t->left != NULL)
        destroy_tree(t->left);
    if (t->right != NULL)
        destroy_tree(t->right);
    free(t);
    t = NULL;
}

void serialize_tree(tree *t, FILE *file){
    if(t == NULL)
        return;
    fwrite((void*)&t->acc, sizeof(account), 1, file);
    serialize_tree(t->left, file);
    serialize_tree(t->right, file);
}

tree* deserialize_tree(FILE *file){
    tree *temp = NULL;
    account acc;
    while(fread(&acc, sizeof(account), 1, file))
        temp = insert_tree(temp, &acc);
    return temp;
}

#endif
```

Демонстрация работы

Client1:

```
$ ./client 8888 alex
adress: tcp://localhost:8888
ask <help> if you don't know what to do
deposit 3000
Operation accepted. Waiting for completion.
Status: OK. Operation completed.
Now your balance is: 9000
transfer nancy 3000
Operation accepted. Waiting for completion.
Status: OK. Operation completed.
Now your balance is: 6000
exit
```

Client2:

```
adress: tcp://localhost:8888
ask <help> if you don't know what to do
balance
Operation accepted. Waiting for completion.
Status: OK. Operation completed.
Now your balance is: 500
balance
Operation accepted. Waiting for completion.
Status: OK. Operation completed.
Now your balance is: 3500
exit
```

Server:

```
$ ./server 8888 accounts.bnk
loading data from accounts.bnk ...
  aaron : 5000
alex : 6000
  edward : 10000
    nancy : 500
adress: tcp://*:8888
alex deposited 3000
nancy requested balance
alex transfered 3000 to nancy
nancy requested balance
^C
saving data to accounts.bnk ...
```

Strace

```
openat(AT_FDCWD, "/proc/self/task/21281/comm", O_RDWR) = 8
write(8, "ZMQbg/1", 7)                  = 7
close(8)                                = 0
...
socket(AF_INET, SOCK_STREAM|SOCK_CLOEXEC, IPPROTO_TCP) = 9
setsockopt(9, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(9, {sa_family=AF_INET, sin_port=htons(8888),
sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(9, 100)                          = 0
getsockname(9, {sa_family=AF_INET, sin_port=htons(8888),
sin_addr=inet_addr("0.0.0.0")}, [128->16]) = 0
write(6, "\1\0\0\0\0\0\0\0", 8)         = 8
```

```
write(8, "\1\0\0\0\0\0\0\0", 8)         = 8
poll([{fd=8, events=POLLIN}], 1, -1)    = 1 ([{fd=8, revents=POLLIN}])
read(8, "\1\0\0\0\0\0\0\0", 8)          = 8
poll([{fd=8, events=POLLIN}], 1, 0)     = 0 (Timeout)
poll([{fd=8, events=POLLIN}], 1, -1)    = 1 ([{fd=8, revents=POLLIN}])
```

Выводы

Очередь сообщений

В случае недостатка ресурсов, мы не можем сразу же обработать посылаемые данные. Тогда поставщик данных ставит их в очередь сообщений на сервере, который будет хранить данные до тех пор, пока клиент не будет готов. И эти данные точно долежат до тех пор, пока мы не освободимся и их не обработаем.

Очередь сообщений предоставляет гарантии, что сообщение будет доставлено независимо от того, что происходит. Очередь сообщений позволяет асинхронно взаимодействовать между слабо связанными компонентами приложения, а также обеспечивает строгую последовательность очереди. Т.е. совершенно точно кто первый пришел, того первым и обработают.

Достоинства и недостатки

Зачастую на низком уровне довольно сложно построить систему взаимодействия отдельных приложений, поэтому это может занять значительное время. Но если потратьить время и правильно разработать систему, будет получена высокая скорость взаимодейтсвия а так же необходимая гибкость настройки для данной конкретной задачи.

Средства высокого уровня по каким-то причинам могут проигрывать в скорости взаимодействия и, к тому же не всегдя обладают достаточной гибкостю для проектирования каких-то нестандартных решений. Но, они в свою очередь, позволяют проектировать подобные системы довольно быстро.

Библиотека же ZMQ является неким компромисом между высоким им низким уровнем. Она вобрала лучшее от обоих сторон. Благодаря этой библиотеке может быть достигнута высокая скорость взаимодействия, и не будет затрачено слишком много времени на разработку.

(В силу отсутствия опыта сам судить не берусь. Все вышесказанное мнения самих разработчиков ZMQ и тех, кто мало-мальски ей пользовался)

Что это такое?

ZeroMQ – библиотека, реализующая очередь сообщений, которая помогает разработчикам создавать распределенные и параллельные приложения.

Данная библиотека, дает разработчику некий более высокий уровень абстракции для работы с сокетами.

Библиотека берет на себя часть забот по буферизации данных, обслуживанию очередей, установлению и восстановлению соединений, и тому подобное.

Благодаря этому разаработчик, вместо того чтобы самостоятельно программировать все вышеперечисленное на низком уровне, берет готовое (по отзывам во многом хорошее решение) и далее работает уже над архитектурой и логикой приложения.

Итого

Как результат была разработана простейшая система серевер-клиент, в какой-то мере моделюрующая банковские операции. В ходе ее разработки приобретены начальные знания и опыт в программировании управлении серверами сообщений, применении отложенной обработки и проектированиия взаимодействия приложений друг с другом. Так же приобретен какой никакой опыт в работе с конкретной библиотекой ZeroMQ. Это совершенно точно полезный опыт, и с подобным, скорее всего, придется столкнуться еще не раз на этом тернистом пути становления программиста.