Лабораторная работа 1

**Синхронизация часов (I)**

* Используйте материал «Синхронизация\_времени».
* Используйте параллельный сервер, предоставленный преподавателем.
* Продемонстрируйте работу параллельного сервера, обслуживающего любые запросы двух клиентов, работающих на разных компьютерах.
* **Схема синхронизации на 1ом этапе**.



* Разработайте программы клиента и UDP-сервера следующей функциональности.
* Клиент периодически шлет серверу запросы следующей структуры.



* Периодичность клиентского запроса ***Tc*** является параметром программы-клиента и задается в тиках, равных **1/1000 сек**.
* Сервер без задержки отвечает клиенту на каждый запрос ответом, имеющим следующую структуру.



* Значение счетчика времени ***Cs*** на сервере – это время работы сервера с момента его запуска в тиках (функция **clock** из **<ctime>** стандартной библиотеки С++).
* Первоначальное значение счетчика времени ***Cc*** на клиенте равно **нулю**. Т.е. при первой отправке запроса текущее значение **curvalue** будет равно **нулю**.
* После получения ответа от сервера клиент корректирует (прибавляет) значение счетчика на величину **correction** (это значение может быть положительны, ноль или отрицательным).
* Задержка на сторонне клиента между запросами моделируется с помощью функции **Sleep**. Параметром этой функции является заданное параметром значение ***Tc***.
* Значения счетчика ***Cc*** увеличивается клиентом после каждой задержки **перед отправкой запроса** на величину задержки ***Tc***. Т.е. **curvalue будет** равно **curvalue + correction + *Tc***.
* При получении запроса, сервер вычисляет значение **correction = *Cs -*** **curvalue**.
* **После отправки ответа** сервер выводит на консоль IP-клиента, номер запроса, отправленную величину **correction**, и среднее значение **correction**.
* Проведите серию из 7 экспериментов (1 сервер, 2 клиента) на основе которого заполняется следующая таблица. Для каждого эксперимента принимается устанавливается значение ***Tc***. В каждом эксперименте клиент должен делать по 10 запросов

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Tc*** | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 |
| ***Max correction*** |  |  |  |  |  |  |  |
| ***Min correction*** |  |  |  |  |  |  |  |
| ***Среднее correction*** |  |  |  |  |  |  |  |

* Оформить отчет о выполнении лабораторной работы.

и на сервере и на клиенте для подсчета коррекции будем использовать не массив коррекций, а вектор. для этого нужно включить библиотеку с вектором:  
#include <vector>

дальше в теле функции main() нужно объявить вектор corrections:  
vector<int> corrections;

теперь можно написать функцию подсчета среднего значения коррекции. для этого находим сумму всех коррекций и делим на их количесвто:  
int calculateAverageCorrection(vector<int> &corrections)

{

int sum = accumulate(corrections.begin(), corrections.end(), 0);

return sum / corrections.size();

}

для работы синхронизации времени сначала нужно настроить сервер. для этого на сервере инициализируем переменную serv. указываем, что сервер будет работать с протоколами TCP/IP (AF\_INET), указываем порт для прослушивания (htons(2000)) и указываем, что сервер может принимать запросы от любого адреса (INADDR\_ANY) :  
SOCKADDR\_IN serv;

serv.sin\_family = AF\_INET;

serv.sin\_port = htons(2000);

serv.sin\_addr.s\_addr = INADDR\_ANY;

при инициализации сокета, нужно указать, что сокет будет работать по протоколу UDP:  
sS = socket(AF\_INET, SOCK\_DGRAM, NULL))

далее связываем сокет с данными, которые инициализировали выше:  
bind(sS, (LPSOCKADDR)&serv, sizeof(serv)

сокет на сервере проинициализирован, теперь можно ожидать подключения от клиентов:  
recvfrom(sS, (char\*)&getsincro, sizeof(getsincro), NULL, (sockaddr\*)&client, &lc);  
  
далее нужно инициализировать работу клиента. для гибкой настройки клиента будут использоваться параметры командной строки, чтобы указать ip адрес сервера, его порт и задержку в тиках:  
 string IP = "127.0.0.1";

int PORT = 2000;

int Tc = 1000;

if (argc != 4)

{

cout << "Должно быть 3 параметра: адрес сервера, порт сервера, значение задержки в тиках" << endl;

cout << "Пример: 192.168.1.3 2000 1000" << endl;

return 0;

}

else

{

IP = argv[1];

PORT = atoi(argv[2]);

Tc = atoi(argv[3]);

}

при запуске программы клиента в параметрах командной строки нужно указать данные строго в следующем порядке: ip адрес сервера, порт, который он прослушивает, количество тиков для задержки между запросами на сервер в 1/1000 сек.

после этого на клиенте инициализируется сокет для работы по UDP:  
cS = socket(AF\_INET, SOCK\_DGRAM, NULL)

далее указывается IP адрес и порт сервера:  
SOCKADDR\_IN serv;

serv.sin\_family = AF\_INET;

serv.sin\_port = htons(PORT);

serv.sin\_addr.s\_addr = inet\_addr(IP.c\_str());

после инициализации всех данных отправляем запрос на сервер:  
sendto(cS, (char\*)&getsincro, sizeof(getsincro), 0, (sockaddr\*)&serv, sizeof(serv));  
  
после обработки запроса, сервер высылает обратно ответ:  
recvfrom(cS, (char\*)&setsincro, sizeof(setsincro), 0, (sockaddr\*)&serv, &lensockaddr);  
  
согласно заданию при первой отправке запроса текущее значение времени равно 0, поэтому при первой отправке текущее значение времени просто приравнивается значению коррекции, которое отправил сервер:  
getsincro.curvalue += setsincro.curvalue;

после первоначальной установки значения времени выполняется запрос, в котором на сервер отправляется текущее значение времени:

sendto(cS, (char\*)&getsincro, sizeof(getsincro), 0, (sockaddr\*)&serv, sizeof(serv));

после того, как данные приходят на сервер, присходит их получение:  
recvfrom(sS, (char\*)&getsincro, sizeof(getsincro), NULL, (sockaddr\*)&client, &lc);

далее сервер получает значение тиков, которое прошло со старта программы:

c = clock();

согласно заданию, сервер вычисляет коррекцию (время сервера - время клиента), и отправляет эти данные обратно клиенту:  
setsincro.correction = c - getsincro.correction;

sendto(sS, (char\*)&setsincro, sizeof(setsincro), 0, (sockaddr\*)&client, sizeof(client));

также на сервере ip адрес клиента из поля (&client)->sin\_addr преобразуется в строку:  
inet\_ntop(AF\_INET, &(reinterpret\_cast<sockaddr\_in\*>(&client)->sin\_addr), clientIP, INET\_ADDRSTRLEN);

и происходит подсчет средней коррекции:  
corrections.push\_back(setsincro.correction);

average = calculateAverageCorrection(corrections);

после этого на клиент приходит ответ от сервера, устанавливается значение времени с учетом коррекции и задержки:

recvfrom(cS, (char\*)&setsincro, sizeof(setsincro), 0, (sockaddr\*)&serv, &lensockaddr);

getsincro.curvalue += setsincro.curvalue + Tc;

после этого на клиенте происходит подсчет минимального и максимального значения коррекции:  
maxcor = (maxcor < setsincro.curvalue) ? setsincro.curvalue : maxcor;

mincor = (mincor > setsincro.curvalue) ? setsincro.curvalue : mincor;

и также добавление текущей коррекции к массиву всех коррекций:  
corrections.push\_back(setsincro.curvalue);

в конце цикла на клиенте происходит искуственная задержка с помощью функции sleep:  
Sleep(Tc);

после того, как проведено 10 попыток синхронизации времени с сервером, происходит подсчет средней коррекции:  
int average = calculateAverageCorrection(corrections);

в данном алгоритме при запуске клиента и при первой отправке запроса на сервер инициализируется время клиента. так как запуск клиента может произойти в любой момент, не обязательно сразу после запуска сервера, то это может повлиять на подсчет средней коррекции на сервере. данная проблема была решена следующим способом:  
клиент генерирует уникальный идентификатор подключения и отправляет его вместе с данными, необходимыми для коррекции времени. сервер проверяет, было ли уже соединение с этим клиентом. если было, то сервер проводит подсчеты, а если нет - то считается, что сервер отправляет клиенту время в качестве коррекции. поэтому это первое соединение не нужно учитывать во время подсчета коррекции. соответственно сервер запоминает клиента и будет учитывать его коррекции в последующих соединениях. далее будет описаны инструменты и алгоритм, с помощью которых это было достигнуто.

на клиенте генерируем уникальное число и записываем его в объект структуры, который будем отправлять на сервер:  
struct GETSINCHRO

{

string cmd;

int curvalue;

long long unique\_id; // уникальный идентификатор для клиента

// будет использоваться в сервере для подсчетов

};

auto now = chrono::system\_clock::now();

getsincro.unique\_id = chrono::duration\_cast<std::chrono::milliseconds>(now.time\_since\_epoch()).count();

было притяно решение использовать данный метод, так как существует ничтожно маленькая вероятность запуска клиента в одну и ту же миллисекунду времени.

сервер принимает это значение, преобразует его в строку и добавляет его к ip адресу клиента. далее проверяет, есть ли полученное значение в уже известных соединениях:  
string client\_id = clientIP;

client\_id += to\_string(getsincro.unique\_id);

if (connected\_clients.find(client\_id) == connected\_clients.end()) {

connected\_clients.insert(client\_id);  
}  
else {  
// подсчет средней коррекции  
}

Результаты проведенных экспериментов:

Лабораторная работа 1

Client 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Tc*** | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 |
| ***Max correction*** | 9 | 9 | 9 | 10 | 11 | 10 | 10 |
| ***Min correction*** | 2 | 1 | 2 | 2 | 3 | 1 | 3 |
| ***Среднее correction*** | 7 | 7 | 7 | 7 | 8 | 8 | 8 |

Client 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Tc*** | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 |
| ***Max correction*** | 10 | 9 | 10 | 10 | 9 | 10 | 11 |
| ***Min correction*** | 3 | 2 | 1 | 2 | 1 | 2 | 3 |
| ***Среднее correction*** | 7 | 7 | 8 | 8 | 8 | 7 | 8 |