**Университет ИТМО**

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.03.04 Информатика и вычислительная техника

Дисциплина «Низкоуровневое программирование»

**Отчет**

По лабораторной работе №3

Вариант 3 (Protocol Buffer)

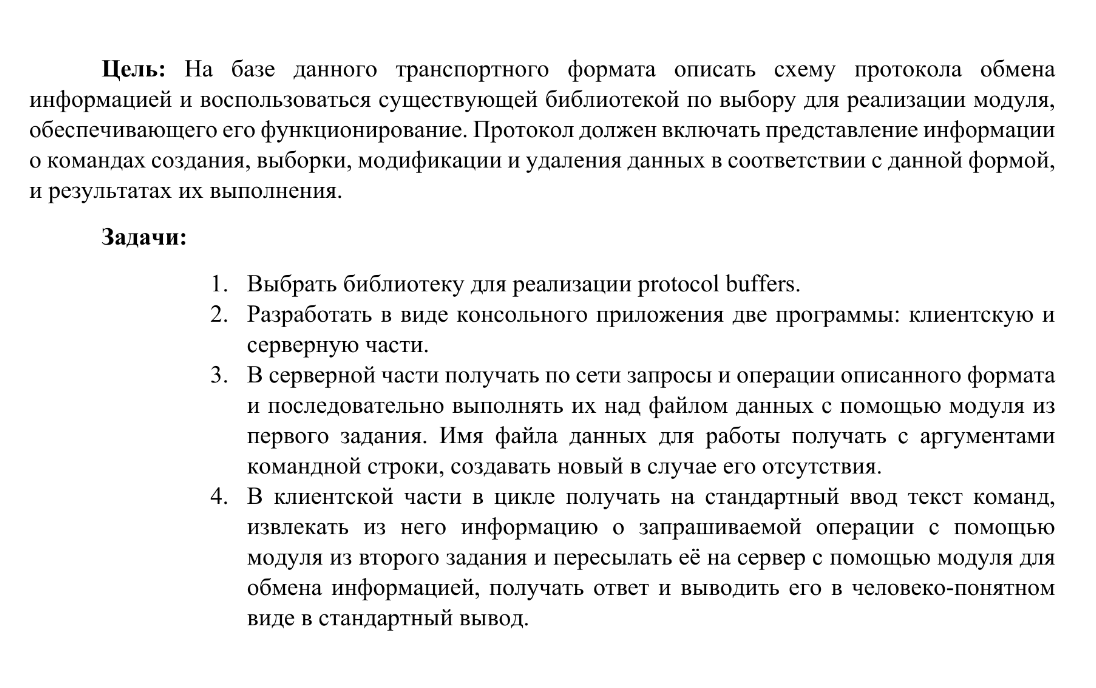
Выполнил:

*Степанов М.А.*

Преподаватель:

*Кореньков Ю. Д.*

Санкт-Петербург, 2023 г.

**Описание работы:**

Программа представляет из себя синтез артефактов полученных в 1 и 2 лабораторных работах (database\_module), (parser\_module), в частности объединение их в две взаимодействующих программы: клиент и сервер. Makefile’ы лабораторных работ были переделаны в файлы cmake и включены в новый модуль в качестве подключаемых библиотек. Для сериализации данных использовался Protocol Buffer, реализуемый библиотекой nanopb (использовалась именно она, потому что было найдено много материалов в интернете + она потребляет меньше оперативной памяти). Сборка конечных артефактов (client, server) осуществляется при помощи cmake. В качестве аргументов серверу передается адрес локальной конечной точки подключения и адрес подключаемого файла, а также имя файла, с которым будет взаимодействовать сервер. В качестве аргументов клиенту передается адрес локальной конечной точки для подключения.

**Аспекты реализации:**

Для работы с protocol buffer необходимо сформировать два файла (message.pb.h, message.pb.c). Для этого необходимо написать файла с расширением .proto в качестве описания протокола.

Message.proto:

syntax = "proto2";  
  
message Attribute {  
 required int32 type = 1;  
 required string name = 2;  
 oneof values {  
 string str = 3;  
 int32 integer = 4;  
 float real = 5;  
 int32 int32ean = 6;  
 }  
}  
  
message Operator {  
 required int32 field = 1;  
 required Attribute value = 2;  
}  
  
message Comparator {  
 required int32 negative = 1;  
 required int32 is\_true = 2;  
 required int32 operation = 3;  
 required Operator op1 = 4;  
 required Operator op2 = 5;  
}  
  
message Filter {  
 required int32 negative = 1;  
 repeated Comparator comp = 2;  
}  
  
message Level {  
 required int32 negative = 1;  
 required int32 any = 2;  
 required int32 parent = 3;  
 required int32 id = 4;  
 repeated Filter filters = 5;  
}  
  
message Query\_tree {  
 required int32 command = 1;  
 repeated Level level = 2;  
 required string name = 3;  
 required int64 value = 4;  
}  
  
message Return\_code {  
 required int32 code = 1;  
}  
  
message Entity {  
 repeated Attribute values = 1;  
}  
  
message Tuple {  
 required Entity data = 1;  
 required int32 id = 2;  
 required int32 parent\_id = 3;  
}  
  
message Collection {  
 repeated Tuple tuples = 1;  
}

В качестве результата работы сервер отправляет либо сущность Return\_code (содержит только код возврата), либо список искомых элементов (Collection). Ответ отправляется пакетами по 1024 байт, имеется ограничение на длину названия аттрибутов в 16 символов (данное ограничение указано в конфигурации и можно изменить переконфигурирова proto файл. Так как Protocol Buffer имеет обратную совместимость, это не должно привести к устареванию данных).

Библиотека не поддерживает удаленный вызов функций, поэтому для связи по сети были использована API ОС.

Client:

while(1) {  
 sockfd = do\_connection(argv[1], hints);  
 printf("client: print your request:\n");  
 code = execute(sockfd);  
 if (!code) {  
 printf("client: successfully executed\n");  
 } else {  
 printf("client: error %ld was occurred\n", code);  
 break;  
 }  
 close(sockfd);  
}

Server:

while (1) {  
 sin\_size = sizeof their\_addr;  
 new\_fd = accept(sockfd, (struct sockaddr \*) &their\_addr, &sin\_size);  
 if (new\_fd == -1) continue;  
  
 inet\_ntop(their\_addr.ss\_family,get\_in\_addr((struct sockaddr \*) &their\_addr),s, sizeof s);  
 printf("server: got connection from %s\n", s);  
  
 if (!fork()) {  
 close(sockfd);  
 do\_connection(new\_fd, argv[2]);  
 close(new\_fd);  
 exit(1);  
 }  
 close(new\_fd);  
}

client.do\_connection:

for(p = serv\_info; p != NULL; p = p->ai\_next) {  
 if ((sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1) {  
 perror("client: socket\n");  
 continue;  
 }  
  
 if (connect(sockfd, p->ai\_addr, p->ai\_addrlen) == -1) {  
 close(sockfd);  
 perror("client: connect\n");  
 continue;  
 }  
  
 break;  
}

server connection:

if (listen(sockfd, BACKLOG) == -1) {  
 perror("server: listen\n");  
 exit(1);  
}

client decode and encode:

ostream = pb\_ostream\_from\_buffer(output\_buffer, sizeof(output\_buffer));  
pb\_encode(&ostream, Query\_tree\_fields, &tree);  
  
if (send(sockfd, output\_buffer, BUFFER\_SIZE, 0) == -1) {  
 perror("client: couldn't send message\n");  
 close(sockfd);  
}  
  
if ((recv(sockfd, input\_buffer, BUFFER\_SIZE, 0)) == -1) {  
 perror("client: couldn't receive message\n");  
 close(sockfd);  
}  
istream = pb\_istream\_from\_buffer(input\_buffer, sizeof(input\_buffer));  
  
switch (tree.command) {  
 case *CRUD\_FIND*:  
 pb\_decode(&istream, Return\_code\_fields, &collection);  
 for(size\_t i = 0; i < collection.tuples\_count; i++){  
 printf("--- TUPLE %d ---\n", collection.tuples[i].id);  
 printf("name: %s\n", collection.tuples[i].data.values[0].values.str);  
 printf("code: %d\n", collection.tuples[i].data.values[0].values.integer);  
 }  
 returnCode.code = 0;break;  
 default:  
 pb\_decode(&istream, Return\_code\_fields, &returnCode); break;  
}

server decode and encode:

if ((recv(new\_fd, input\_buffer, BUFFER\_SIZE, 0)) == -1) {  
 perror("server: couldn't receive message\n");  
}  
  
istream = pb\_istream\_from\_buffer(input\_buffer, sizeof(input\_buffer));  
pb\_decode(&istream, Query\_tree\_fields, &tree);  
  
struct result\_list\_tuple \*list;  
  
uint64\_t code = parse\_request(filename, tree, &list);  
  
if (tree.command == *CRUD\_FIND*) {  
 collection = get\_collection(list);  
 ostream = pb\_ostream\_from\_buffer(output\_buffer, sizeof(output\_buffer));  
 pb\_encode(&ostream, Collection\_fields, &collection);  
} else {  
 returnCode.code = code;  
 ostream = pb\_ostream\_from\_buffer(output\_buffer, sizeof(output\_buffer));  
 pb\_encode(&ostream, Return\_code\_fields, &returnCode);  
}  
  
if (send(new\_fd, output\_buffer, BUFFER\_SIZE, 0) == -1) {  
 perror("server: couldn't send message\n");  
}

**Результаты:**

Вывод сервера не многословен (если файл не инциализирован происходит тихая инциализация как в 1 лабе):

/home/mike/LLP/LowLevelPrograming3/cmake-build-debug/LLP3 127.0.0.1 simple.data

server: waiting for connections…

server: got connection from 127.0.0.1

server: got connection from 127.0.0.1

server: got connection from 127.0.0.1

В клиенте в зависимости от запроса разный вывод:

Прим 1:

client: connecting to 127.0.0.1

client: print your request:

?/1/\*

--- TUPLE 4 ---

name: mike

code: 123

client: successfully executed

Прим 2:

client: connecting to 127.0.0.1

client: print your request:

+/123[]/

client: error 1 was occurred

Прим 3:

client: connecting to 127.0.0.1

client: print your request:

+/123

client: successfully executed

**Выводы:**

* Изучил, неизвестную мне до этого, библиотеку nanopb и способ передачи данных по сети — Protocol Buffer. Узнал о существующих плюсах данного способа — ускорение передачи данных, уменьшение используемой памяти и минусах — абсолютной не читаемости данных человеком (и малое количество библиотек для C — их намного больше для C++)
* Научился интегрировать модули cmake, применяя библиотеки и несколько целей сборки (гораздо удобнее, чем Makefile, как мне показалось).
* Был реализован протокол передачи данных при помощи Protocol Buffer, а также передача этих данных по сети.