네트워크소프트웨어설계(F136-1) 팀프로젝트 설계 최종보고서

‘헝그리맨(HUNGRY-MEN)’

<목 차>

소프트웨어학과 201621654 박민지

소프트웨어학과 201720768 김수영

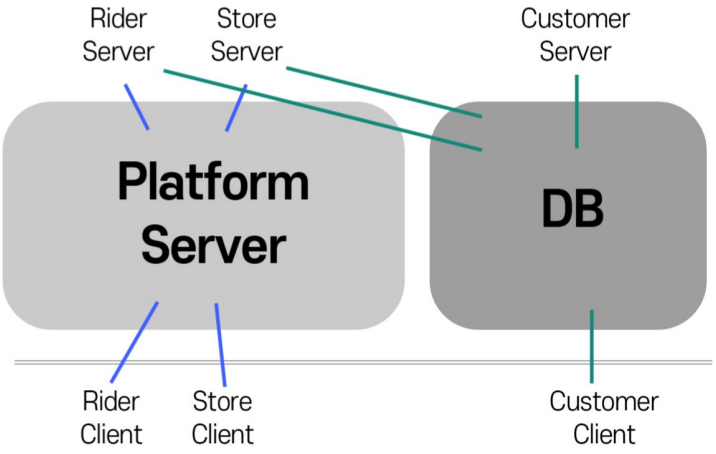
소프트웨어학과 201821225 구교현

1. 목적 및 설계 배경

본 보고서는 기존 상용 서비스를 벤치마킹하여 플랫폼 기반 네트워크 서비스 소프트웨어 설계 프로젝트의 상세 구현방법을 담았다. 본 설계를 통해 플랫폼에 대한 이해와 전반적인 네트워크 이해를 상향시키고자 한다. 먼저 플랫폼을 이해하고자 두 가지의 유형으로 나눴다. 첫 번째 유형은 한정되지 않은 사용자를 대상으로 하는 Maker 플랫폼 형태이다. 이 플랫폼은 다양하고 많은 종류의 Value들을 이용한다는 것이 특징이다. 두 번째 유형은 Maker 플랫폼과 다르게 사용자들의 one-to-one 방식으로 이뤄져 있는 Exchange 플랫폼 형태이다. 본 팀 프로젝트에서는 Exchange 유형의 플랫폼을 사용하고자 한다. 설계 제안 발표 때 Exchange 플랫폼 형태의 사례인 “에어 비앤비”, “배달의 민족”, “아마존” 플랫폼 서버를 선택하였는데 실생활 속에서 접근성이 뛰어나고 익숙한 플랫폼인 배달의 민족을 벤치마킹하여 설계했다. 배달의 민족을 벤치마킹하여 설계한 ‘헝그리맨(Hungry men)’이라는 플랫폼은 “배고플 때 가장 먼저 찾게 되는 플랫폼”이라는 슬로건 아래에 구현 계획을 세웠다. 구현 계획에 따라 설계한 자세한 내용은 본문에서 언급할 것인데 서비스 순서대로 설명하고자 한다.먼저 Store-Customer 간의 음식을 주문하는 서비스에 대해서 설명할것이고, 두번째로 전체적인 플랫폼 서버의 구조에 대한 설명과, Store-Rider간의 주문한 데이터를 기반으로 한 배달 상태를 공유하는 서비스, Store-Rider간 광고 연결 서비스를 설명할 것이다.

2. 시스템 구조도 및 설계 환경

Window10 환경과 VisualStudio를 통해 UDP 기반으로 프로그램 구현했다.



[사진1. 헝그리맨 시스템 구조도]

상단의 이미지는 구현한 ‘헝그리맨’의 시스템 구조도다. Store, Customer, Rider 각각 서로의 데이터 베이스를 참고하고 연관성 있는 서비스가 다양한 서버를 통해 구현되고 있는 것도 플랫폼이라고 설명할 수 있지만 좀 더 구체적인 플랫폼을 구축하고자 했다. 따라서 Store, Customer, Rider 서버 이외에 플랫폼 서버를 따로 두어 이 플랫폼 서버가 서로를 연관지을 수 있도록 했다. 설계 초반 Store와 Rider의 연결관계에 집중했기 때문에 Client는 플랫폼 서버와 엮지 않았다. 비록 이번 설계에서는 연결시키지 않았지만 플랫폼의 특징인 확장성을 살려 Client 또한 추후에 플랫폼 서버와 연결할 수 있을 것이다.

3. 프로그램 설계 #1 (플랫폼 서버)

3.1. 프로토콜

1) Syntax

```
typedef struct Destination {
        int type;
        int number;
        SOCKADDR_IN addr;
}Destination;
```

- Platform Server에서 Store,Rider의 Client, Server를 구분하는 구조체
- type field : 가리키는 위치가 서버인지 클라이언트 인지 구분하는 용도 (Client : type = 0, Server : type = 1)
- number field : type이 1일 경우 (서버일 경우) 어떤 서버인지 구분해주는 구분자.(Store: number = 1, Rider: number = 2)

```
typedef struct PlatformProtocol {
        Destination start;
        Destination end;
        int flag;
        char data[950];
}PlatformProtocol;
```

- Platform Server에서 플랫폼을 통해 통신하는 Server, Client의 경로를 설정하는 구조체
- start field : 통신의 시작 지점을 나타냄, end field : 통신의 도착 지점을 나타냄
- flag : 해당 통신의 성격을 나타냄(rider와 store의 flag를 표현한 것이다.)

(1) 플랫폼 서버를 이용하는 모든 통신 개체들은 위의 PlatformProtocol이 제시하는 형태를 갖추어야 한다. PlatformProtocol를 통해서 플랫폼에 접속한 개체들이 누구와 연결되어야 할지 알 수 있다.

(2) PlatformProtocol를 따른다면 start와 end가 둘 다 NAT 프로토콜을 거치는 client가 아니라면 서로간의 통신이 가능해진다. 기존의 client server 구조로는 server는 자신이 담당한 client만 통신하고 client는 자신을 담당하는 server와 통신했다면 이 플랫폼에서는 통신 주체들이 특정 주체하고만 통신해야하는 제약이 없어진다. 가령 Rider server와 Store server와의 통신이라던지 Store client와 Rider server와의 통신이 PlatformProtocol을 잘 정의하면 쓰레드와 같은 별다른 조치 없이 가능해진다.

(3) Destination의 number field의 경우 플랫폼에서 서비스를 지원하는 서버의 수가 늘어날 때마다 가능한 인자 값의 수도 증가한다. 만약 Customer가 서버에 제대로 안착한다면 number 3은 Customer 서버를 가리키게 될 것이다.

(4) 시작 지점과 도착 지점의 type 값이 0인데 시작 지점과 도착 지점의 number 값이 같다면 이는 서버가 자기 자신으로 통신을 요청한 경우이다. 이 경우 단순히 loopback 처리를 해주는 것이 아니라 data 필드의 query 문을 실행시켜주는 api 역할을 수행한다. 발표에서 말했듯 지금은 단순히 쿼리 문을 실행시켜주는 정도이지만 인터페이스가 정의되고 보안과 권한이 확정되며 기능이 추가된다면 멋진 api가 될 수 있을 것이다.

2) Semantic

(1) retval = recvfrom(sock, buf, BUFSIZE, 0, (SOCKADDR)&clientaddr, &addrlen);*
플랫폼 서버로 들어오는 데이터들을 받는다.

(2-1) retval = sendto(sendsock, (char)platformbuf, sizeof(PlatformProtocol), 0, (SOCKADDR*)&destination, sizeof(SOCKADDR_IN));*
분기 처리 이후 해당하는 목적지로 패킷을 보낸다.

3) 프로토콜 구현

플랫폼 서버의 연결 기능은 크게 서버와 서버, 클라이언트와 서버로 나누어진다. 클라이언트와 클라이언트는 클라이언트의 NAT 프로토콜이 동작하는 한 서버에서 해당 클라이언트의 정확한 IP를 확정할 수 없어 지원하지 못하였다. 또 서버가 먼저 클라이언트로 접근하는 것 또한 클라이언트가 private IP를 사용하므로 지원하지 못한다. 다만 client 가 먼저 서버에 접근한다면 NAT 프로토콜에 의해 해당 client의 외부의 접근이 private IP로 변환되므로 client가 먼저 접근하는 방식의 통신은 가능하다.

구현은 생각보다 단순하다. PlatformProtocol의 start와 end의 속성을 확인하여 Destination 구조체의 SOCKADDR_IN addr에 값을 넣어주고 end가 가리키는 위치로 패킷을 전달하면 된다. 우선 Destination의 type이 0인 경우 서버를 의미하고 플랫폼은 number를 통해 어떤 서버로 접근했는지 알 수 있다. 해당 number에 대응되는 SOCKADDR_IN은 이미 플랫폼 서버에 등록되어 있기 때문에 해당 값을 대입하면 된다. type이 1인 경우에서 먼저 client가 server로 보낼 때 recvfrom을 통해 받은 SOCKADDR 값을 넣어준다. 이 값은 server에서 client로 보낼 때 까지 PlatformProtocol 구조체의 start 또는 end 구조체의 내에 저장되어 진다. 이런 식으로 sockaddr_in을 통해 통신 정보를 기록해두어 원하는 곳으로 패킷이 전달되도록 한다.

여기서 쓰인 PlatformProtocol은 플랫폼 내의 통신 주체들의 프로토콜을 감싸는 프로토콜이다. java의 인터페이스처럼 PlatformProtocol이 제시하는 형태를 만족한다면 이 프로토콜을 사용할 수 있다. char data 부분에 다르게 정의된 프로토콜 기반으로 만들어진 데이터가 전송된다.

4) 동작 결과

플랫폼 서버 자체만으로는 동작을 보여줄 수 없다. 뒤에 이어서 설명하는 store와 rider의 동작에 기반이 될 뿐이다.

4. 프로그램설계 (Customer <-> Store | 주문 서비스)

4.1 프로토콜

1) Syntax

Store,Customer,Rider의 정보를 저장하고, 유동적으로 사용할 수 있도록 mysql 프로그램을 이용하였다.VisualStudio의 콘솔창과 mysql의 데이터베이스와 연결하기 위해 하단과 같은 코드를 이용하였다.

```
MYSQL* conn; //mysql과의 커넥션을 잡는데 지속적으로 사용되는 변수
MYSQL_RES* res;  //쿼리문에 대한 result값을 받는 위치변수
MYSQL_ROW row;   //쿼리문에 대한 실제 데이터값이 들어있는 변수

const char* server = "localhost";  //서버의 경로
const char* user = "root"; //mysql로그인 아이디.
const char* password = "****"; /* set me first */   //password
```

사전에 명령프롬프트를 켜고 'create database netsof;'라는 명령어를 통해 데이터베이스를 생성하였고, 'create table'이라는 명령어를 통해 'store','customer','rider','ordering','Advertisement' 5개의 테이블을 생성하여 각 정보에 맞는 데이터 유형에 맞는 칼럼들을 생성했다. 데이터베이스 테이블 관련 정보는 하단과 같다.

- **Store DB**

Field	Type	Null	KEY	Default
_no	int(11)	NO	PRI	NULL
id	varchar(45)	NO		NULL
password	varchar(45)	NO		NULL
storename	varchar(45)	NO		NULL
address	varchar(45)	NO		NULL
category	varchar(45)	NO		NULL

먼저 스토어의 데이터베이스 구성은 상단에 보이는 표와 같다. 첫 칼럼에 있는 '_no' 변수는 모든 테이블 안에 있는 value로써 row 번호를 통해 효과적으로 데이터를 찾기 위해서 넣었다. auto increasement로

데이터가 insert 될 때 마다 값이 증가할 수 있도록 해두었다. store에 필요한 정보인 id, password 등을 문자열 형태로 담았다.

- **Customer DB**

Field	Type	Null	KEY	Default
_no	int(11)	NO	PRI	NULL
id	varchar(45)	NO		NULL
password	varchar(45)	NO		NULL
address	varchar(45)	NO		NULL

상단에 있는 Customer 데이터베이스 테이블의 칼럼들은 위에서 설명했던 Store와 동일한 내용이므로 설명을 생략한다.

- **Ordering DB**

Field	Type	Null	KEY	Default
_no	int(11)	NO	PRI	NULL
storename	varchar(45)	NO		NULL
menuname	varchar(45)	NO		NULL
ridername	varchar(45)	YES		NULL
storestatus	int(11)	YES		NULL
riderstatus	int(11)	YES		NULL
time_stamp	datetime	YES		CURRENT

상단에 있는 표는 주문 접수와 관련된 데이터베이스 칼럼을 담은 테이블이다. 위에서 언급한 Store 데이터베이스 테이블과 같은 맥락으로 칼럼을 구성하였지만, time_stamp라는 변수를 통해 주문이 들어오는 즉시 시간을 자동적으로 기록할 수 있게끔하여서 주문 시간을 store,customer,rider에게 디스플레이 할 수 있게끔 했다. 추가적인 부분으로 ridername, storestatus, riderstatus 값을 null로 설정할 수 있게끔하여서 주문이 들어와도 처음엔 NULL 상태였다가 주문이 store, rider와 매칭이 되는 경우에 값을 변경할 수 있게끔 했다.

2) Semantics

(1) retval = recvfrom(sock, clientinfo.recvlogin, sizeof(clientinfo), 0, (SOCKADDR*)&clientaddr, &addrlen);

: 자신에게 들어오는 데이터를 받는다. 서버는 클라이언트에게서 온 데이터를, 클라이언트는 서버에서 온 데이터를 받는다.

(2) sendto(sock, orderinfo.category, sizeof(orderinfo.category), 0, (SOCKADDR*)&clientaddr, sizeof(clientaddr))

: 자신이 원하는 데이터를 상대방에게 전송해준다. 클라이언트가 서버에게 전송할때는 orderinfo라는 구조체를 따로 둬서 orderinfo 구조체 안에 회원가입, 로그인, 주문 등에 필요한 정보를 담았다. 구조체 전체를 전송하는 것이 아닌 구조체 안에 있는 변수를 필요에 따라 하나씩 전송하는 semantics다.

(3) mysql_query(conn, "select * from store");

: 원하는 데이터베이스에게 접근을 하는 semantics다. 위에 있는 코드를 통해 select *from store 라는 명령어를 이용하여 store 테이블에 있는 데이터에 대해 접근할 수 있다.

(4) sprintf(query, "select _no from customer where id=₩"%s₩" and password=₩"%s₩"", s.id, s.pw);
mysql_query(conn,query);

: (3)과 동일한 semantics다. 하지만, 차이점은 변수를 담아 문자열 형태로 변환시켜주기 위해 sprintf 함수를 이용해서 전송했다는 것이다. char형 변수 query를 설정하여 명령어에 맞는 데이터(예를 들어 %s는 s.id인데 id에 대한 정보를 넘기는 것이다.)를 문자열 형태로 바꿔 원하는 데이터베이스에 데이터를 insert 시켜준다.

4.2 프로토콜 구현

1) Customer Client & Customer Server

주문은 Customer의 가장 핵심적인 기능이다. 회원가입과 로그인에 대한 부분은 3.프로그램 설계 #1(플랫폼)에서 설명했으므로 긴 설명은 생략한다. Customer Client 콘솔창을 통해 로그인 혹은 회원가입이 완료된다면, Customer Server는 카테고리를 선택할 수 있게끔 카테고리 종류를 sendto 함수를 이용하여 전송한다. Customer가 recvfrom 함수를 통해 카테고리에 대한 정보를 받고, 원하는 카테고리의 정보를 담아서 전달해준다. 많은 음식 카테고리가 존재하지만, 편리하게 검색하기 위해서

“한식”,”양식”,”중식”,”일식” 네 부분으로 나눴다. 원하는 카테고리가 Client측에서 도착하면 Server측에서 그 카테고리에 맞는 Store의 정보를 Customer 측에 전달해준다. 이 때 Store의 정보를 받아오기 위해서 Customer Server는 Store의 데이터베이스를 이용하는데 이는 2)Store Client & Store Server 부분에서 설명할 예정이다.

```
while (row = mysql_fetch_row(res)) {
        /*카테고리1) 한식*/
        if (orderinfo.selectnum == 1) {
            if (strcmp(row[5], KOREA_FOOD) == 0) {
                sprintf(storearr[storecount].storeinfo, "  %d)%s₩n", storecount + 1, row[3]);
                storecount += 1;
            }
            else
                continue;
        }

          (중략)
}
```

위의 코드는 Store 데이터베이스에 접근하여 데이터를 search하는 일을 한다. row[5] 즉, 카테고리 영역이 미리 #define 해놓은 KOREA_FOOD가 일치하면 storecount를 count하며 카테고리 내에 상점이 있음을 알리고, sotrearr이라는 구조체 배열 내에 storeinfo 문자열에 정보를 저장한다. 만약 맞는 정보가 없어서 storecount==0 인경우, 즉 Store의 데이터베이스에 등록된 정보에서 만약 맞는 카테고리 내에 등록된 상점이 없다면 “카테고리 내 주문 가능한 상점이 존재하지 않습니다”라는 메시지를 sendto 하고 다시 카테고리를 물어본다. 미리 데이터베이스에 등록된 정보에서 카테고리 내에 등록된 상점을 count하고 한꺼번에 모아서 sendto 시키기 위하여 storearr라는 구조체 배열을 만들었다.

상점이름을 받은 client는 상점의 이름을 sendto해준다. server는 그 정보를 받고 실제 store 데이터베이스 내에 있는 store의 이름인지 확인하고 맞다면 “메뉴 이름 : “을 sendto 해주어 Client 가 메뉴를 선택할 수 있게끔 해주었고, 만약 store의 이름이 잘못 입력되었다면 다시 상점이름을 선택하라는 메시지를 send하게끔 해주었다. 잘못 입력했을 경우, Server가 sendto 보내는 만큼 Client도 recvfrom 받아야하기 때문에 while문을 이용했는데 서로 일치할 때를 알기 위해 아래와 같은 flag를 설정했다.

```
  retval = recvfrom(sock,recvinfo.choosemenu,sizeof(recvinfo.choosemenu), 0,
(SOCKADDR*)&peeraddr, &addrlen);
  if (retval == SOCKET_ERROR) {
```

```
    err_display("recvfrom()");
    continue;   }
 recvinfo.choosemenu[retval] = '₩0';
 if (strcmp(recvinfo.choosemenu, "0") == 0) {
        printf("입력하신 상점이 없습니다!₩n");
        continue;
   }
```

만약 Server가 0을 전송하면 Client는 printf를 통해 “입력하신 상점이 없습니다”라는 오류 메시지를 출력하고 continue를 통해 다시 데이터를 recv 받을 준비를 한다. 입력한 상점과 recvfrom 받은 상점이름이 일치하면 Client는 메뉴 선택과 주소를 입력하고 다시 서버에게 sendto한다. Client의 sendto를 통해 order에 대한 정보를 받은 Server는 아래와 같은 코드로 Ordering 데이터 베이스에 접근하여 데이터를 insert 시켜준다.

```
 sprintf(query, "insert into ordering
(storename,menuname,ridername,storestatus,riderstatus,clientaddress) values
(₩"%s₩",₩"%s₩",₩"%s₩",%d,%d,₩"%s₩");", orderinfo.recvstore, orderinfo.menuname,
NULL, 0, 0,orderinfo.addr);

  mysql_query(conn, query);
```

Ordering 데이터베이스 테이블에서도 언급했듯이, riderstatus와 storestatus는 Customer의 관할이 아니기 때문에 0으로 설정을 해준다. 그리고 이 주문에 대해 Rider의 매칭이 이뤄지지 않은 상태이기 때문에 NULL값으로 설정한 것이다. 주문이 제대로 insert되었다면, Server는 다시 주문에 대한 정보를 Client에게 전달해준다. Client는 주문 정보를 확인하고 프로그램을 종료시켜준다.

2) Store Client & Store Server

주문 단계에서 Store가 해주어야하는 일은 Customer Server가 Store 데이터 베이스에 접근할 수 있도록 Store 데이터베이스에 데이터를 insert 시켜주는 일이다. 로그인은 이미 등록된 데이터베이스의 데이터에 대하여 접근하는 것이기 때문에 주문하는 부분에서는 회원가입 코드만 보면 된다.

```
case  2:
   response=(AdminProtocol*)malloc(sizeof(AdminProtocol));
   registerprocess((AdminProtocol*)buffer, (AdminProtocol*)response,conn);
```

```
    retval=sendto(scok,(char*)response, sizeof(AdminProtocol),0,(SOCKADDR*)&clientaddr,
sizeof(clientaddr));
```

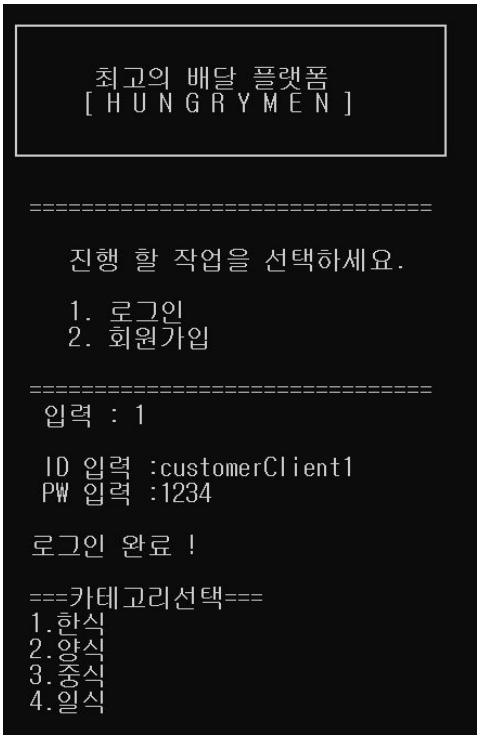
회원가입과 관련된 코드는 아래와 같다. Store Client는 접속할 때 원하는 일을 선택할 수 있는데 1.로그인 2. 회원가입 3. 이다. 따라서 회원가입의 flag인 2를 Server에게 전송해준다. flag를 받은 Server는 switch 문을 통해서 case 2번으로 가게 되는데 case 2번에서 registerprocess라는 함수를 이용한다.

```
void registerprocess(AdminProtocol* loginbuf, AdminProtocol* response, MYSQL* conn)
{
   memset(response, NULL, sizeof(response));
   if (checkWithLogin(conn, loginbuf) != -1) response->result = 0;
   else
   {
      InsertStoreInfo(conn, loginbuf);
      response->store_index = checkWithLogin(conn, loginbuf);
      response->result = 1;
   }
   response->flag = 2;
   response->start.type = 1;
   response->end.type = 0;
   response->start.number = 1;
   response->end.addr = loginbuf->start.addr;
}
```

위는 회원가입시 사용하는 registerprocess 함수 코드이다. 여기에서 start.type, end.type, start.number, end.addr는 플랫폼 서버를 통해 다른 연락 매체와 통신하기 위한 프로토콜이므로 현재 인증 관련 기능하고는 무관하다. 간략히 설명하자면 데이터를 전송하는 쪽은 서버에 상점이므로 type은 1, number는 1이고 받는 쪽은 클라이언트이니 type은 0에 end.addr에 해당 client의 sock_addr_in 값을 넣는다. checkWithLogin을 통해 로그인이 가능하다면 이는 이미 있는 아이디와 비밀번호 이므로 회원가입이 불가능하다. 만약 checkWithLogin이 false를 리턴한다면 이는 회원이 아직 없다는 의미이므로 회원가입을 진행한다.

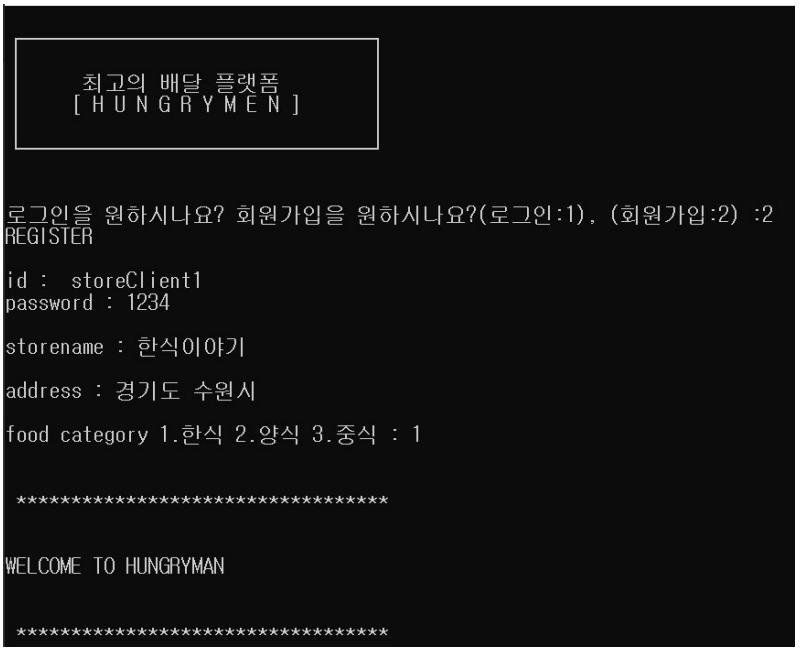
4.3 동작 결과

(1) 주문하기 전, Store와 Customer가 맨처음 하는 일은 로그인 혹은 회원가입을 하는 것이다. 첫 번째는 Customer에서 로그인하는 모습이다.



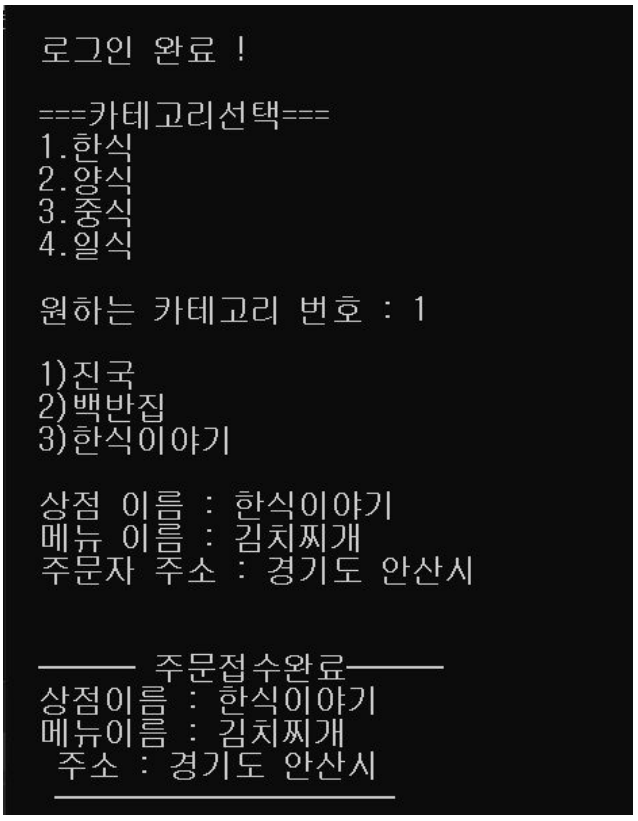
	_no	id	password
▶	3	cust...	1234
*	NULL	NULL	NULL

(2) 아래 사진은 Store가 회원가입을 통해 데이터를 데이터베이스에 insert 시키는 것이다.



	_no	id	password	storename	address	category
▶	11	storeClient2	1234	진국	경기도 안양시	한식
	12	storeClient3	1234	백반집	서울특별시	한식
	13	storeClient4	1234	황제짬뽕	경기도 의왕시	중식
	14	storeClient1	1234	한식이야기	경기도 수원시	한식

(3) 주문이 잘 들어가서 주문 내역을 출력하고, 데이터베이스에 잘 들어간 모습



5. 프로그램 설계 #2(Store <-> Rider | 배달 서비스)

5.1 프로토콜

*각 client와 server가 직접 통신을 하지 않고, Store Server와 Rider Server가 Platform Server에 내재 돼 있는 방식으로 syntax, semantics은 모두 Platform Server로 송수신된다.

*따라서 syntax와 semantics가 명시적으로는 순서가 분명하지만, 실제 통신 순서와 다음 설명과는 semantics부분에서 일치하지 않고, 2번 프로토콜을 포함한다.

1) syntax

* Store Client가 전달하는 syntax 구분 flag

* Rider Client가 전달하는 syntax 구분 flag

* flag = 3; 주문 정보 요청
* flag = 4: 광고 등록 요청
* flag = 5: 주문 정보 변경 요청

* flag = 3; 주문 정보 요청
* flag = 4; 광고 정보 요청
* flag = 5; 배달 과정 정보 요청
* flag = 6; 해당 주문 승낙
* flag = 7; 광고 정보 승낙
* flag = 8; 특정 주문 상태 변화

(1) Store Client가 Platform Server에게 주문과 관련된 명령을 전달하는 syntax

```
typedef struct orderingInfoList
{
        Destination start;
        Destination end;
        int flag;
        int store_index;
        int whole_row;
        OrderingInfo Info[5];
}OrderingInfoList;
```

다음과 같은 OrderingInfoList구조체에 Destination start부분은 client임을 명시하는 type 0를 Destination end부분은 Store Server와 연결되기 위한 type 1과 number 1 데이터를 넣는다. store_index에는 이전에 로그인|회원가입 을 하면서 받아온 Store 데이터베이스상 자신의 _no 값을 넣어준다. whole_row는 Server에게 전달 받을 list의 길이로 값을 넣지 않고, 전체 OrderingList의 최대 길이는 5로 지정한다. 구조체 크기만큼 잘라서 문자열 형태로 변환후 Platform Server에 전달되는 syntax이다.

(2) Store Server가 Platform Server에 주문 list를 전달하는 syntax

Ordeing 데이터베이스에 관련된 내용은 4.프로그램설계 #2에서 언급했으므로 언급을 생략하겠다. (1)에서 언급한 OrderingInfoList구조체에 OrderingInfo 구조체 배열을 참조해서, 각각의 데이터를 해당 Store 과 관련된 Ordering 데이터베이스에서 query문을 통해 가져와 데이터를 저장한다.flag와 Destination 정보 Server에서 Server로 전달하는 것을 명시하는 start,end type = 1, number = 1를 지정하고 OrderingInfoList를 문자열 형태로 변환후 구조체 크기만큼 잘라서 전송하는 syntax이다.

(3) Rider Client가 Platform Server에게 자신의 정보를 전달하는 syntax

- **Rider DB**

Field	Type	Null	KEY	Default
_no	int(11)	NO	PRI	NULL
id	varchar(45)	NO		NULL
password	varchar(45)	NO		NULL
ridername	varchar(45)	NO		NULL

상단에 있는 Rider 데이터베이스 테이블의 칼럼들은 1번 프로그램 설계(주문 서비스)에서 언급한 Store 데이터베이스 테이블의 칼럼과 유사하므로 설명을 생략한다.

이전의 Destination 구조체의 데이터를 (1)과 같은 형식으로 지정하고, 정보를 Rider 데이터베이스에 저장할 field기반으로 데이터를 담은 전체, AdminProtocol 구조체 형식의 데이터를 문자열로 변환해 구조체 크기만큼 전달하는 syntax

(4) Rider Client가 Platform Server에게 배달과 관련된 명령을 전달하는 syntax
syntax (1)과 동일, OrderingInfoList와 동일한 역할을 하는 ListProtocol구조체에 Destination start부분은 client임을 명시하는 type 0를 Destination end부분은 Store Server와 연결되기 위한 type 1과 number 2 데이터를 넣는다. flag은 일전에 지정한 데이터로 지정하고, numofstore은 Server에게 전달 받을 list의 길이로 값을 넣지 않고, 전체 상점 Info 배열의 최대 길이는 5로 지정한다. 구조체 크기만큼 잘라서 문자열 형태로 변환후 Platform Server에 전달되는 syntax이다.

(5) Rider Server가 Platform Server에 배달 list를 전달하는 syntax
(1)에서 언급한 OrderingInfoList구조체에 OrderingInfo 구조체 배열을 참조해서, 전체 주문 list 데이터와 관련된 Store 데이터베이스에서 가져온 데이터를 가져온다. flag 와 Destination정보를 Server에서 Server로 전달하는 start,end type = 1, number 2로 지정하고, ListProtocol 구조체 형식을 문자열로 변환하여 전달하는 syntax.

(6) 이후로 syntax는 Rider Client와 Store Client가 list의 해당 ordering list를 선택하는 데이터를 담은 syntax들이다.

2) semantics(반환값 : 실패시 error (-1), 성공시 수신,송신 둘다 전달받고 전달하는 syntax의 길이)

Store (주문 list확인)

(1)retval = sendto(sock, (char*)orderList, sizeof(OrderingInfoList), 0,(SOCKADDR*)&serveraddr, sizeof(serveraddr)); *(Store Client)*

Store Client은 로그인||회원가입을 완료하고, 특정 주문 list를 확인해서 조리 완료를 누르기 위해서 먼저 주문 list를 요청하는 semantics이다. OrderingInfoList syntax를 전송하고, 이 정보를 전달 받는 소켓 주소 포인터 즉 목적지 (Platform Server)의 ip주소는 "127.0.0.1"와 port number는 8600으로 지정했다. 앞으로 모든 상점 라이더 client 와 server는 해당 소켓 주소 포인터로 데이터를 전송한다.

(2) retval = recvfrom(sock, buffer, BUFFERSIZE, 0,(SOCKADDR*)&clientaddr, &addrlen);*(Store Server)*

Platform Server를 통해 Store Client에게 받은 list요청 데이터를 받고 받은 데이터를 OrderingInfoList형태로 변환해서 flag에 따라 분류한다.해당 flag = 3, udp 통신의 서버 부분으로 client_addr socket 주소를 통해 다음 과 같은 정보를 받는 semantics.

(3) retval = sendto(sock, (char*)command_response, sizeof(OrderingInfoList), 0,(SOCKADDR*)&clientaddr, sizeof(clientaddr)); *(Store Server)*

해당 Store Client의 정보데이터를 기반으로 이에 해당하는 주문 list를 OrderingInfoList의 구조체 형식으로 Platform Server에게 전달하는 semantics.

(4) retval = recvfrom(sock, buffer, BUFFERSIZE, 0,(SOCKADDR*)&peeraddr, &addrlen);*(Store Client)*

이전에 지정한 peer address를 통해서 OrderingInfoList형태의 syntax를 받아와 list를 client에게 display해주기 위해서 Platform Server를 통해 데이터를 받아오는 semantics.

Rider(로그인 || 회원가입)

로그인 || 회원가입 기능은 부가적인 기능이므로 syntax로만 설명을 하겠다.

Rider(주문 list확인 + 픽업 || 접수 명령 전달)

해당 flag = 3,5와 syntax가 달라지고 전체적인 semantics구성 Store과 동일

*(1)****retval = sendto(sock, (char*)request, sizeof(ListProtocol), 0, (SOCKADDR*)&serveraddr, sizeof(serveraddr));*** *(Rider Client)*

(2) ***retval = recvfrom(sock, buf, BUFSIZE, 0,(SOCKADDR*)&clientaddr, &addrlen);*** *(Rider Server)*

(3) ***retval = sendto(sock, (char*)listresponse, sizeof(ListProtocol), 0, (SOCKADDR*)&clientaddr, sizeof(clientaddr));*** *(Rider Server)*

(4) ***retval = recvfrom(sock, buf, BUFSIZE, 0, (SOCKADDR*)&peeraddr, &addrlen);*** *(Rider Client)*
픽업 || 접수 명령 전달
*(5)****retval = sendto(sock, (char*)senddata, sizeof(AcceptProtocol), 0,(SOCKADDR*)&serveraddr, sizeof(serveraddr));*** *(RiderClient)*

Platform Server를 대상으로 AcceptProtocol 구조체 형식의 syntax를 flag 6,7,8과 list의 itemid를 지정해서 전달하는 semantics.

*(6)****retval = recvfrom(sock, buf, BUFSIZE, 0,(SOCKADDR*)&clientaddr, &addrlen);*** *(Rider Server)*

전달 받는 방식은 해당 프로그램 설계의 semantics들과 동일

*(7)****retval = sendto(sock, (char*)acceptprotocol, sizeof(ListProtocol), 0,(SOCKADDR*)&clientaddr, sizeof(clientaddr));*** *(Rider Server)*

이전에 지정한 client addr 소켓 주소 포인터로 Ordering 데이터베이스 업데이트 요청 완료 syntax를 전달하는 semantics.

*(8)****retval = recvfrom(sock, buf, BUFSIZE, 0,(SOCKADDR*)&peeraddr, &addrlen);*** *(Rider Client)*

(7)결과를 반환해서 업데이트가 완료 되었는지, 오류가 생겼는지 확인하는 semantics.

<u>Store(조리 완료 명령 전달)</u>
해당 flag = 5와 syntax가 달라지고 전체적인 semantics구성 Rider와 동일하다.
(1) ***retval = sendto(sock, (char*)statuscommand, sizeof(AcceptProtocol), 0,(SOCKADDR*)&serveraddr, sizeof(serveraddr));*** *(Store Client)*
(2) ***retval = recvfrom(sock, buffer, BUFFERSIZE, 0,(SOCKADDR*)&clientaddr, &addrlen);*** *(Store Server)*
(3) ***retval = sendto(sock, (char*)change_command_response, sizeof(AcceptProtocol), 0, (SOCKADDR*)&clientaddr, sizeof(clientaddr));*** *(Store Server)*
(4) r***etval = recvfrom(sock, buffer, BUFFERSIZE, 0,(SOCKADDR*)&peeraddr, &addrlen);*** *(Store Client)*

5.2 프로토콜 구현

1) Rider Client & Rider Server 구현

전체적으로 Rider server에서 먼저 Platform 서버와 통신 할 수 있는 환경을 만들어주고, 앞서 설명한것 처럼 syntax를 semantics순서에 따라 송 수신을 한다. 이때 모든 송수신하는 데이터에 flag parameter를 넣어서, 여러개의 Rider Client가 한번에 Server에 접근할 수 있게 만든다.

```
ListProtocol* request = (ListProtocol*)malloc(sizeof(ListProtocol));
request->start.type = 0;
request->end.type = 1;
request->end.number = 2;
switch (whatdo) {
        case 1:
                request->flag = 3;
                retval = sendto(sock, (char*)request, sizeof(ListProtocol), 0,
                                (SOCKADDR*)&serveraddr, sizeof(serveraddr));
                if (retval == SOCKET_ERROR) {
                        err_display((char*)"sendto()");
                        continue;
                        }
                free(request);
                break;
          ...
          case 3:  request->flag = 5;
          ...
          ...
AcceptProtocol* senddata = (AcceptProtocol*)malloc(sizeof(AcceptProtocol));
senddata->userid = identity;
senddata->start.type = 0;
senddata->end.type = 1;
senddata->end.number = 2;
        switch (whatdo) {
                case 1:
                        senddata->flag = 6;
                        senddata->itemid = (returnlist->info[choicelist - 1]).no;
                        break;
                    ....
                  case 3: senddata->flag = 8;
```

Rider client에서 사용하는 hungrymen의 배달 서비스 기능은 주문을 접수하고, 배달을 완료할 수 있는 기능으로, 이 두개의 Rider client에서 플랫폼 서버에 전송할 수 있는 기능 명령을 flag별로 나누어 Rider Server에 간접적으로 전송을 하기 위해서 먼저 두개의 기능 중 무엇을 Rider Client가 이용할지 입력을 받게 한다. 입력받은 int형 변수를 기준으로 switch()문을 이용해서 앞서 설명한것 처럼 먼저 ordering

데이터베이스와 관련된 list와 상점과 관련된 list를 제공받아야 하기 때문에, 처음부터 ListProtocol 구조체 형식으로 전송하고 flag는 해당 명령대로 나눠서 입력한다.
Rider Client는 list를받아 parsing한 이후에 순서대로 display된 화면을 제공받을 수 있게 print형식으로 이 display를 구현을 한 이후에 어떤 주문을 배달을 완료를 할지 주문 접수를 할지 다시 업데이트 요청을 하는 syntax를 이번에는 AcceptProtocol 구조체 형식으로 Platform server에 sendto()를 한다.
모든 Rider Client가 원하는 과정이 다 이루어지면 다시 기능 선택 메시지를 출력해주어서 프로그램 서비스 사용을 계속할 수 있게 만든다.

```
case 3: case 4: case 5:
        listresponse = (ListProtocol*)malloc(sizeof(ListProtocol));
        listprocess((ListProtocol*)buf, (ListProtocol*)listresponse, conn);
        // 결과를 보낸다.
        retval = sendto(sock, (char*)listresponse, sizeof(ListProtocol), 0,
                                (SOCKADDR*)&clientaddr, sizeof(clientaddr));
        if (retval == SOCKET_ERROR) {
                        err_display((char*)"sendto()");
                        continue;
                        }
        break;
case 6: case 7: case 8:
        acceptprotocol = (AcceptProtocol*)malloc(sizeof(AcceptProtocol));
        acceptprocess((AcceptProtocol*)buf, (AcceptProtocol*)acceptprotocol, conn);
        retval = sendto(sock, (char*)acceptprotocol, sizeof(ListProtocol), 0,
                                (SOCKADDR*)&clientaddr, sizeof(clientaddr));
        if (retval == SOCKET_ERROR) {
                        err_display((char*)"sendto()");
                        continue;
                        }
        break;
```

List 요청 syntax를 전달받은 Rider Server는 그대로 Char 형 배열로 recvfrom()를 한 이후에, Rider Client에서 전송할때 결정했던 flag parameter에 있는 값으로 분류해서 데이터베이스와 관련된 함수들을 실행한다. Rider Client에게 list를 제공하기 위해서 선정한 flag들,3,5에 해당하면 listprocess()를 실행해 각 명령에 해당하는 select, mysql query 문을 실행한다. 가져온 data를 새로운 ListProtocol 구조체 형식의 syntax에 저장을 하고 이 데이터를 Rider Client에 sendto()를 해준다. 같은 방식으로 Rider Client가 ordering 데이터베이스 의 store status, rider status 를 업데이트 시켜주라는 요청을 수신하면 AcceptProtocol 구조체 형식으로 변환, flag를 6,8 별로 분류하여 acceptProcess()를 실행시키고 , 이 함수

내에서 update, mysql query 문을 실행한다. 각각 업데이트의 성공 여부를 다시 새로운 acceptProtocol 구조체 형식의 syntax로 구성해 sendto()를 해준다.

2) Store Client & Store Server 구현

Rider Server와 동일하게 Store server에서 먼저 Platform 서버와 통신할 수 있는 환경을 만든다.이후에 앞서 설명한것 처럼 syntax를 semantics순서에 따라 송 수신을 한다. 이때 모든 송수신하는 데이터에 flag parameter를 넣어서, Rider Client와 마찬가지로 여러개의 Customer Client가 한번에 Server에 접근할 수 있게 만든다.

```
switch (command)
{
        case 1:

                orderList->flag = 3;
                orderList->store_index = result_store_index;
                orderList->start.type = 0;
                orderList->end.type = 1;
                orderList->end.number = 1;
                   …
}
…
switch (commandFromServer->flag)
{
        case 1:
                displayWholeOrderingList((OrderingInfoList*)buffer);
                  printf("₩n라이더가 연결된 조리 완료할 메뉴를 리스트에서 선택해주세요 : ");
                int listnum;
                scanf("%d", &listnum);
                statuscommand->flag = 5;
                statuscommand->itemid = listnum - 1;
                statuscommand->store_index = result_store_index;
                statuscommand->statusid = 1;
                statuscommand->start.type = 0;
                statuscommand->end.type = 1;
                statuscommand->end.number = 1;
…
}
```

Store Client에서 hungrymen Platform 을 통해 사용하는 배달 서비스의 기능은 Rider가 연결된 주문을 Rider가 픽업할 수 있는 상태로 변경하는 것이다. 조리 완료를 할 수 있는 기능은 flag를 3으로 지정해 이 정보를 OrderingInfoList의 구조체 형식으로 syntax를 구성하고 sendto()를 통해 Platform Server에 명령 데이터를 송신한다. 명령 데이터를 송신하면 무조건 주문 list를 recvfrom()하고 다음 명령문을 입력받아야 하기 때문에 전체 Store Client가 배달 서비스 뿐만아니라 다른 서비스를 이용할때 제공받는 기능들을 다시 실행하기 전에 전체 기능 switch()문 바깥에 조건문을 추가해, recvfrom()를 실행하게 구현했다. 해당 주문 list들을 수신 받은 이후에는 displayWholeOrderingList()함수를 통해 list를 순서대로 print문 을 통해서 Store Client에게 제시를 해준다. 같은 조건문 안에서, 특정 list안의 주문 정보를 변경하기 위해서 list의 어떤 위치의 정보를 바꿀껀지 선택을 하게 하고 이 정보를 store Server가 update시키게 하기 위해서 sendto()로 데이터를 보낸다. 이때 flag는 5를 가진다. 이후 연속적으로 recvfrom()를 통해 update 성공 여부를 platform server에게 전달 받아 display 해준다.

```
FlagProtocol* command = (FlagProtocol*)buffer;
AdminProtocol* response;
OrderingInfoList* command_response;
AcceptProtocol* change_command_response;
AdvertisementProtocol* advertisement_command_response;

    switch (command->flag)
    {
case 3:

        command_response = (OrderingInfoList*)malloc(sizeof(OrderingInfoList));
        getWholeOrderingInfo(conn, (OrderingInfoList*)buffer,
(OrderingInfoList*)command_response);

        command_response->flag = 1;
        ...
        break;
case 5:
        change_command_response = (AcceptProtocol*)malloc(sizeof(AcceptProtocol));
        changeOrderingStatus(conn, (AcceptProtocol*)buffer,
(AcceptProtocol*)change_command_response);
         ....
        break;
}
```

Store Server는 List요청을 문자열 형태로 recvfrom()한 이후에 switch문으로 분리해 각 process함수를 실행한다. 먼저 list요청에 대해서는 flag 3으로 분리해 전체 주문 list를 얻어오는 getWholeOrderingInfo()함수를 실행한다. 상점과 관련되어 있는 주문 list를 가져와야 하기 때문에 이 함수 내에서 getStoreInfo()함수를 실행해, 해당 storename를 가져오고 이후에 storename과 관련이 있는 모든 주문 list를 가져오는 select, mysql query 문을 실행한다. 각 데이터는 다시 새롭게 선언한 OrderingInfoList 구조체 syntax에 저장하고 sendto()를 통해 platform server에 전달한다. 주문 list에 대하여 상점의 status를 바꾸는 명령을 Store Client가 AcceptProtocol 구조체 syntax 형태로 sendto()를 하고 platform server를 통해 데이터가 수신되면 이 데이터를 list 요청과 같은 위치에서 recvfrom()하고 flag를 5로 분류한다. Store Server는 changeOrderingStatus() 함수를 통해서 update, mysql query 문을 실행한다. recvfrom()을 통해 받아온 AcceptProtocol와 같은 syntax형태로 업데이트 성공 여부를 저장해 Store Client에 전송하기 위한 sendto()를 실행한다.

“앞서 언급한 것과 같이, 모든 sendto(), recvfrom()은 platform server와의 통신에서 쓰이는 semantics로 각각 Destination 구조체의 데이터 값을 해당 명령의 목적지, 출발지와 Client인지 Server인지에 맞게 numbering을 해서 넣어준다.”

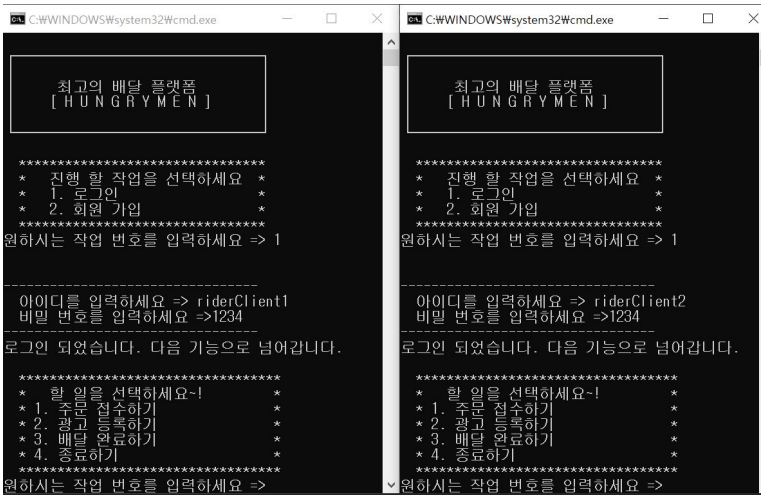
5.3 동작 결과

1) 동작 환경

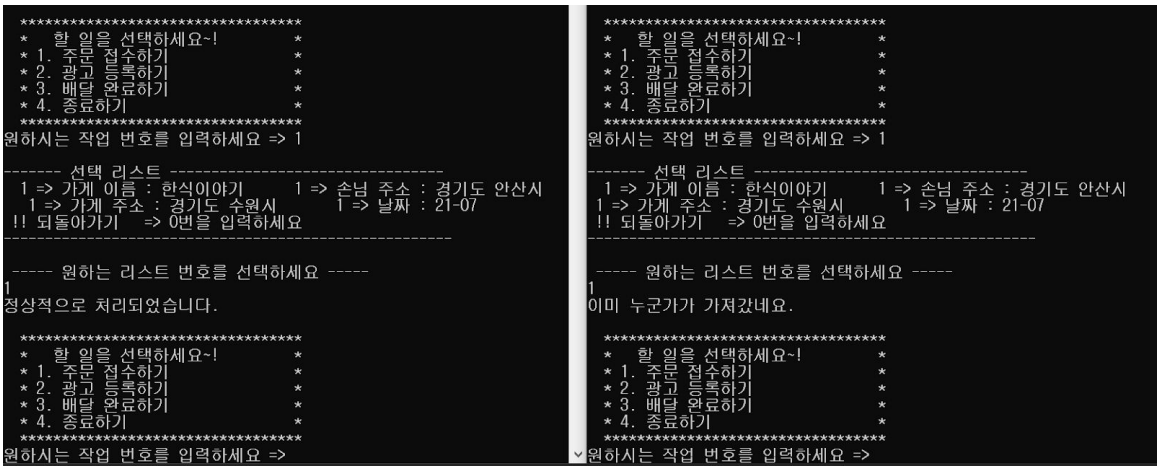
- Rider Server의 portnum : 8400, ip address : 127.0.0.1
- Store Server의 portnum : 8700, ip address : 127.0.0.1

2) 동작 결과

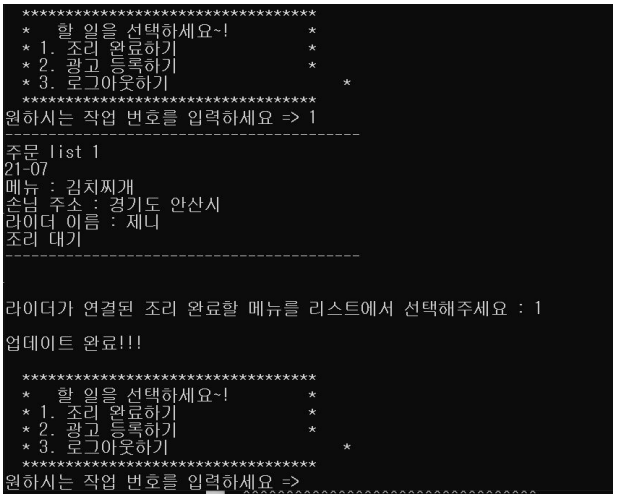
1. Store Client에서 이전에 로그인, 회원가입을 완료한 상태이다.
2. 두개의 Rider Client가 로그인, 회원가입을 완료한 상태로 접수 기능을 선택한다.



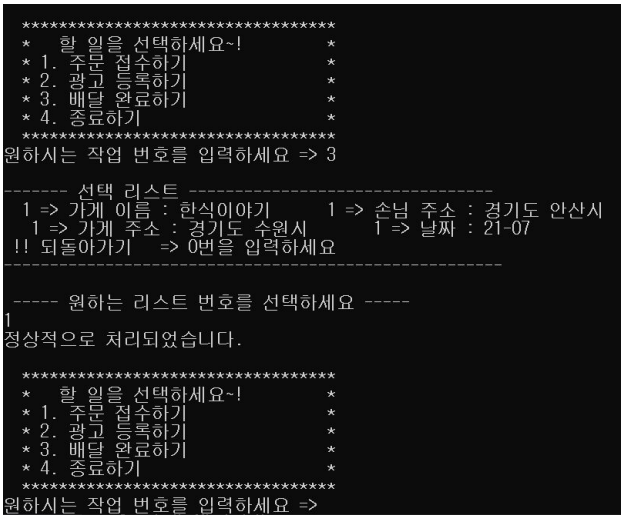
3. Customer Client가 주문한 list를 조회해서 주문 접수하기를 두명의 RiderClient에서 시도하고, Rider Client 하나는 주문 접수 성공 다른 하나는 주문 접수를 실패한 화면이다.



4. Rider Client가 연결된 이후에 Store Client는 주문 list를 조회해서 조리 완료를 할 주문을 선택해서, update를 해준다.



5. Rider Client는 offline으로 Store의 주문 요리를 픽업한다음 배달을 완료한다.



6. 프로그램 설계 #3(Store <-> Rider | 광고 서비스)

6.1 프로토콜

1) Syntax

광고서비스에서 사용되는 Store 데이터베이스에 대한 설명은 1번 프로그램 설계(주문 서비스)에서 언급했고, Rider 데이터베이스에 대한 설명은 2번 프로그램 설계(배달 서비스)에서 언급했기 때문에 이에 대한 설명을 생략한다.

- **Advertisement DB**

Field	Type	Null	KEY	Default
_no	int(11)	NO	PRI	NULL
storename	varchar(45)	NO		NULL
ridername	varchar(45)	YES		NULL

상단에 있는 표는 광고 데이터베이스 테이블이다. 앞서 추가적인 요소에 rider와 store간의 광고 연결을 차별화된 점으로 구성하였기 때문에 덧붙인 테이블이다. 만약 광고를 하길 원하는 store가 있다면 store client 쪽에서 광고에 대한 요청을 하고 그 요청에 대한 부분이 데이터베이스로 inset된다. 그 광고를 하여 수수료에 대한 이익을 원하는 rider는 자신의 이름을 insert 시켜주면 매칭이 되고 그 광고를 하길 원하는 rider가 없다면 NULL상태를 유지한다.

2) Semantic

(1) 상점의 광고 등록

retval = sendto(sock, (char*)advertisementRegister, sizeof(AdvertisementProtocol), 0, (SOCKADDR*)&serveraddr, sizeof(serveraddr));

StoreClient에서 StoreServer로 AdvertisementProtocol 구조체에 flag에 4 값을 넣은 후 전송한다.

retval = recvfrom(sock, buffer, BUFFERSIZE, 0, (SOCKADDR*)&clientaddr, &addrlen);

Store Server에서 AdvertisementProtocol에 flag 값이 4, store_index 값이 해당 store의 아이디인 데이터를 전송 받아

(2) 배달원의 광고 신청

retval = sendto(sock, (char*)request, sizeof(ListProtocol), 0, (SOCKADDR*)&serveraddr, sizeof (serveraddr));

Rider Client에서 ListProtocol 구조체에 flag 값에 4 를 넣어서 서버에 요청한다.

retval = recvfrom(sock, buf, BUFSIZE, 0,(SOCKADDR*)&clientaddr, &addrlen);

서버는 이를 입력 받아서 flag 값을 통해 구분해서 처리한다. 이후 서버에서 클라이언트로 답장하는 방식도 위와 동일하다.

retval = sendto(sock, response, sizeof(AcceptProtocol), 0,(SOCKADDR*)&clientaddr, sizeof (clientaddr));

리스트 중에서 번호를 선택한 후 선택된 리스트의 번호와 flag에 7값을 대입한 AcceptProtocol 구조체를 서버로 보낸다.

retval = recvfrom(sock, buf, BUFSIZE, 0,(SOCKADDR*)&clientaddr, &addrlen);

서버는 이를 입력 받아서 flag 값을 통해 구분해서 처리한다. 이후 서버에서 클라이언트로 답장하는 방식도 위와 동일하다.

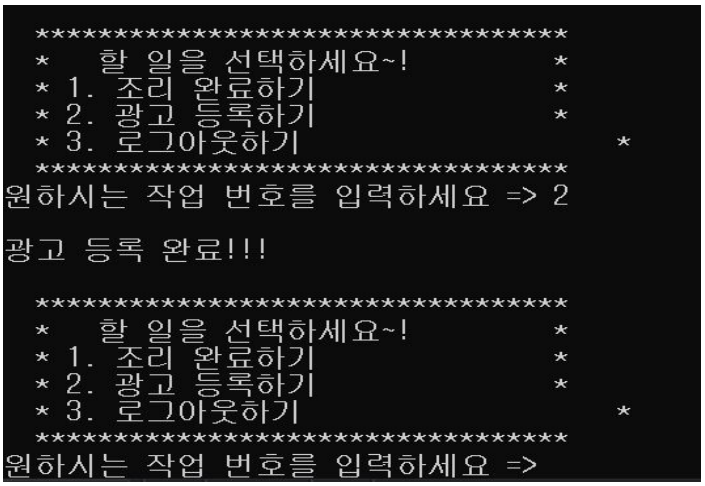
6.2 프로토콜 구현

광고 등록 서비스의 경우 위에서 주문 서비스를 다룰 때와 마찬가지로 Rider Server와 Rider Client 간의 작동과 Store Server와 Store Client간의 작동으로 이루어진다. 먼저 광고 서비스를 시작하기 위해서는 Store가 자신의 광고를 등록하여야 한다. StoreClient에서 advertisementRegister 구조체에서 flag 값을 4로 지정하고 storeindex를 자신의 _no 값으로 지정한 후 StoreServer로 보낸다. StoreServer에서는 해당 구조체의 flag 값을 확인하고 그 값이 4이면 insertAdvertisementInfo 함수를 실행한다. 해당 함수를 통해 상점은 자신의 상점을 광고할 수 있게 된다.

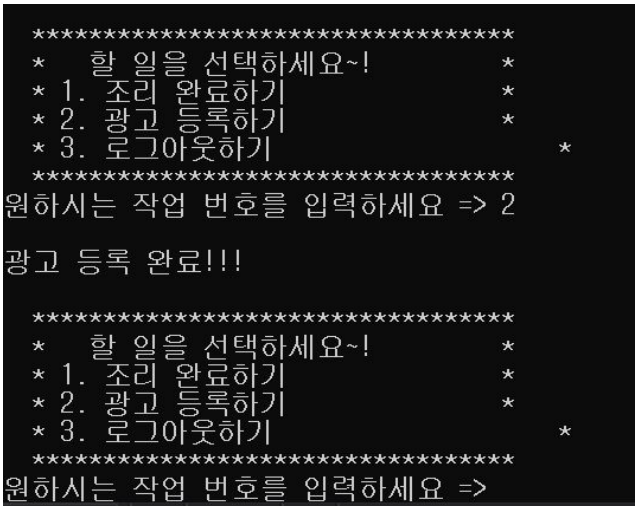
Rider의 경우 먼저 자신이 광고 서비스를 수행할 광고를 선택해야 하므로 ListProtocol 구조체에 광고 리스트를 받을 수 있는 정보를 넣어서 RiderServer로 보낸다. RiderServer에서는 listprocess 함수를 통해 리스트 정보를 데이터베이스에서 꺼내온 뒤 이를 RiderClient로 보낸다. 리스트를 받은 RiderClient는 Server에게 받은 리스트를 보여준 뒤에 Client에게 받은 리스트 중 일부를 선택하게 한다. 선택한 리스트의 _no 값을 AcceptProtocol에 담아 Server로 보내면 서버는 해당 구조체를 입력 받아 키 값이 같은 데이터베이스의 튜플에 null로 차있는 Ridername에 광고를 신청한 Rider name을 넣어준다. 정확한 프로토콜은 코드의 AcceptProtocol를 보면 알 수 있다.

6.3 동작 결과

1. 상점에서 자기자신을 광고에 등록하는 모습이다.



2. 라이더에서 광고 리스트를 받아 광고를 등록하는 모습이다



7. 프로그램 내역

CustomerClient.cpp
CustomerServer.cpp
PlatformServer.cpp
RiderClient.cpp
RiderServer.cpp
StoreServer.cpp
storeClient.cpp

데이터베이스에 관련된 부분은 아래와 같이 insert 시켜놓고 컴파일하면 구현하는데 편리할 것이다.

store DB

INSERT INTO `sys`.`store` (`id`, `password`, `storename`, `address`, `category`) VALUES ('storeClient1', '1234', '한식이야기', '경기도 수원시', '한식');
INSERT INTO `sys`.`store` (`id`, `password`, `storename`, `address`, `category`) VALUES ('storeClient2', '1234', '진국', '경기도 안양시', '한식');
INSERT INTO `sys`.`store` (`id`, `password`, `storename`, `address`, `category`) VALUES ('storeClient3', '1234', '백반집', '서울특별시', '한식');
INSERT INTO `sys`.`store` (`id`, `password`, `storename`, `address`, `category`) VALUES ('storeClient4', '1234', '황제짬뽕', '경기도 의왕시', '중식');

==========

rider DB

INSERT INTO `sys`.`rider` (`id`, `password`, `ridername`) VALUES ('riderClient1', '1234', '제니');
INSERT INTO `sys`.`rider` (`id`, `password`, `ridername`) VALUES ('riderClient2', '1234', '로제');

==========

customer DB

INSERT INTO `sys`.`customer` (`id`, `password`) VALUES ('customerClient1', '1234');

컴파일시, PlatformServer 먼저 컴파일 후 RiderClient,RiderServer, StoreClient,StoreClient를 실행시켜 각 편의에 맞게 사용하면 된다. 플랫폼 서버와 연결되지 않는 CustomerClient,Customer Server는 따로 컴파일 시키거나 함께 컴파일 시켜도 된다.

db와 관련된 server의 환경 구성 & 전체 프로젝트 환경 구성

프로젝트 속성 : x64(플랫폼)
포함 디렉터리 : C:₩Program Files₩MySQL₩MySQL Server 8.0₩include
라이브러리 디렉터리 : C:₩Program Files₩MySQL₩MySQL Server 8.0₩lib
링커 : ws2_32.lib 사용
#pragma comment(lib, "libmySQL.lib") 를 코드에 필수적으로 포함해야한다.