

# 8팀 분산시스템 템 프로젝트 보고서

## 1. Team member information and role

이태호: 조장 50%

정상원: 조원 50%

## 2. Project goal

### A. Goal

- 이 미들웨어는 서버와 클라이언트 앱들이 편하게 서로 통신할 수 있게 만드는 미드웨어입니다. 원래는 차량끼리 서로 정보를 공유하기 위해서 만드는 미들웨어로 만드려고 했으나, 만들다보니 다른 용도로도 사용이 가능합니다. 서버와 클라이언트 앱은 소켓, tcp 연결을 걱정하지 않고, CarEvent 객체를 생성해서 클라이언트나 서버에게 보낼 수 있습니다. 이 미드웨어의 주목적은 클라이언트랑 서버 앱끼리 정보를 주고 받는 것을 편하게 하는 것입니다.

### B. Project Development Environment

- Eclipse IDE를 쓰고 Java Socket을 이용하여 networking을 할 계획입니다. External library로는 org.json을 사용합니다. Server의 인프라는 Aws ec2를 썼습니다.

## 3. Project Design

### A. Details of System Design

공통 클래스:

#### 1) CarEvent

- 이 객체는 어플리케이션과 미들웨어가 서로 주고 받는 메세지 역할을 하는 객체입니다. 이 미들웨어를 쓰는 어플리케이션은 이 안에 정보를 저장하고, 미들웨어는 이 객체를 다른 기기에 있는 미들웨어한테 보내고, 메세지를 받은 미들웨어는 어플리케이션한테 넘겨줍니다. 이 객체의 이름이 CarEvent이나, 사실 자동차와 아무런 관련이 없습니다. ClientSide에서 Server로 보낼 때 HEADER에 "ID" (대문자 주의)가 없으면 미들웨어는 보내지 않습니다!

Field:

JsonObject jsonObj - 보내야 될 메세지를 json 형태로 보관하고 있습니다.

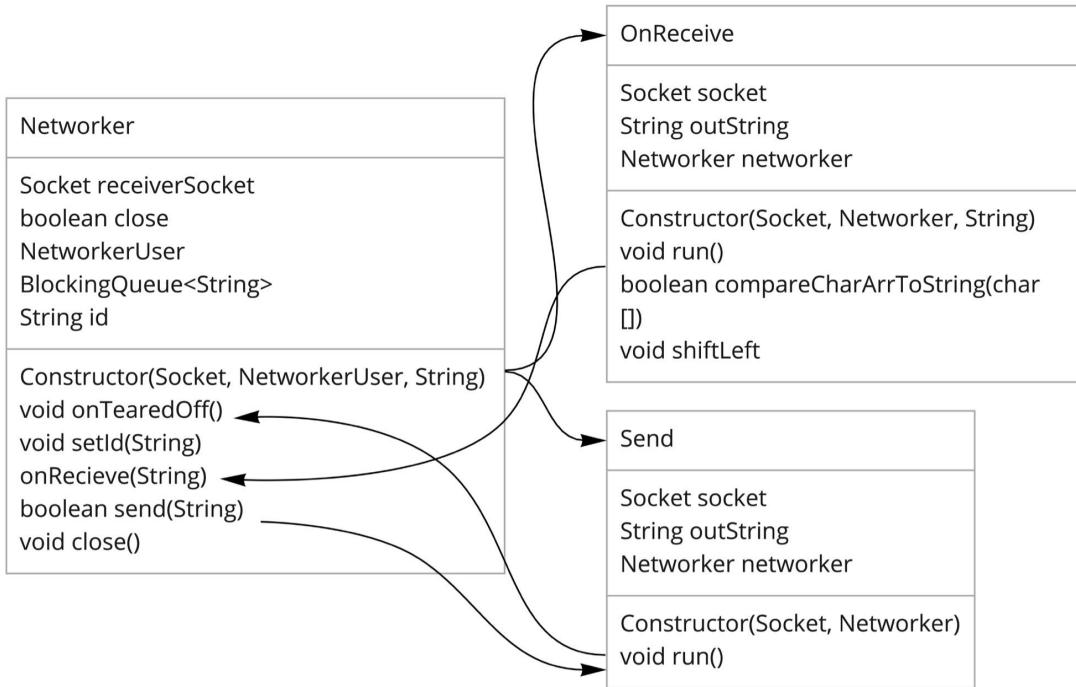
Methods:

- Constructor() - JsonObject를 생성 후 객체 안에 HEADER와 BODY를 생성 한다
- Constructor(JsonObject) - argument로 받은 JsonObject를 jsonObj에 보관
- isEmpty() - jsonObj이 null이 아닌지 확인
- putBody(String key, String value) - Body 부분에 key-pair 형식으로 저장
- putHeader(String key, String value) - Header 부분에 key-pair 형식으로 저장
- getBody(String key) - Body 부분에서 key를 사용해서 값을 가져간다. String 값을 리턴
- getHeader(String key) - Header 부분에서 key를 사용해서 값을 가져간다. String 값을 리턴
- toString() - jsonObj을 toString()한 값 리턴

#### 2) LinkedListQueue

- 이름 그대로 `LinkedList`로 만든 `queue`입니다. 프로젝트 초기에는 많이 사용했지만 지금은 `BlockingQueue`를 더 많이 사용하고 있습니다.
  - `add()`로 `enqueue`를 할 수 있고, `remove()`로 `dequeue`를 할 수 있습니다.

### 3) Networker



- 이 객체는 **String**(보낼 메세지)를 받고, **socket**을 사용해서 다른 기기에게 메세지를 보냅니다. 메세지를 받으면, **NetworkerUser** 객체(constructor에서 argument로 받는다).에게 보냅니다. 이 객체를 사용하려면, **NetworkerUser** 인터페이스를 implement한 객체를 constructor의 argument로 넘겨야 됩니다.
  - **Field:**
    - **Socket receiverSocket** - networker가 다른기기와 통신할때 쓰는 소켓입니다
    - **boolean close** - networker가 닫쳤는지 알려주는 field입니다.
    - **NetworkerUser** - networker가 메세지를 받을때 이 객체에게 넘겨줍니다.
    - **BlockingQueue<String> sendStrings** - 보낼 메세지를 여기에 저장합니다.
    - **String id** - clientId를 저장하는 용도로 쓰입니다. 서버에서 주로 쓰는 field입니다.
  - **Methods:**
    - **Constructor(Socket sock, NetworkerUser networkerUser, String protocolEnd)** - 사용할 소켓과 NetworkerUser 객체, 그리고 메세지의 끝을 알리는 String을 받습니다.
    - **onTearedOff** - socket과의 연결이 끊겼을때 호출하는 함수입니다. 이 함수는 SendThread 객체가 socket의 연결이 끊겼을때 호출하는 함수입니다. 이 함수는 다시 networkerUser의 onTearedOff를 호출합니다. 이때, networker는 보내지 못한 메세지를 다시 networkerUser에게 보내줍니다.
    - **onRecieve** - 메세지를 받을때 networkerUser의 onRecieve를 호출해서 메세지를 보내줍니다.

- `send()` - `sendString`에 보낼 메세지를 enqueue합니다.
- `close` - `socket`을 닫습니다.

#### 4) Send

- 이 객체는 `networker` 객체의 `sendStrings`를 주기적으로 체크를 해서, 메세지가 있으면 `socket`을 사용해서 보내는 함수입니다. 이 클래스는 `Thread class`의 subtype입니다. `socket`과의 연결이 끊기면 `networker`의 `onTearedOff()`함수를 호출합니다.

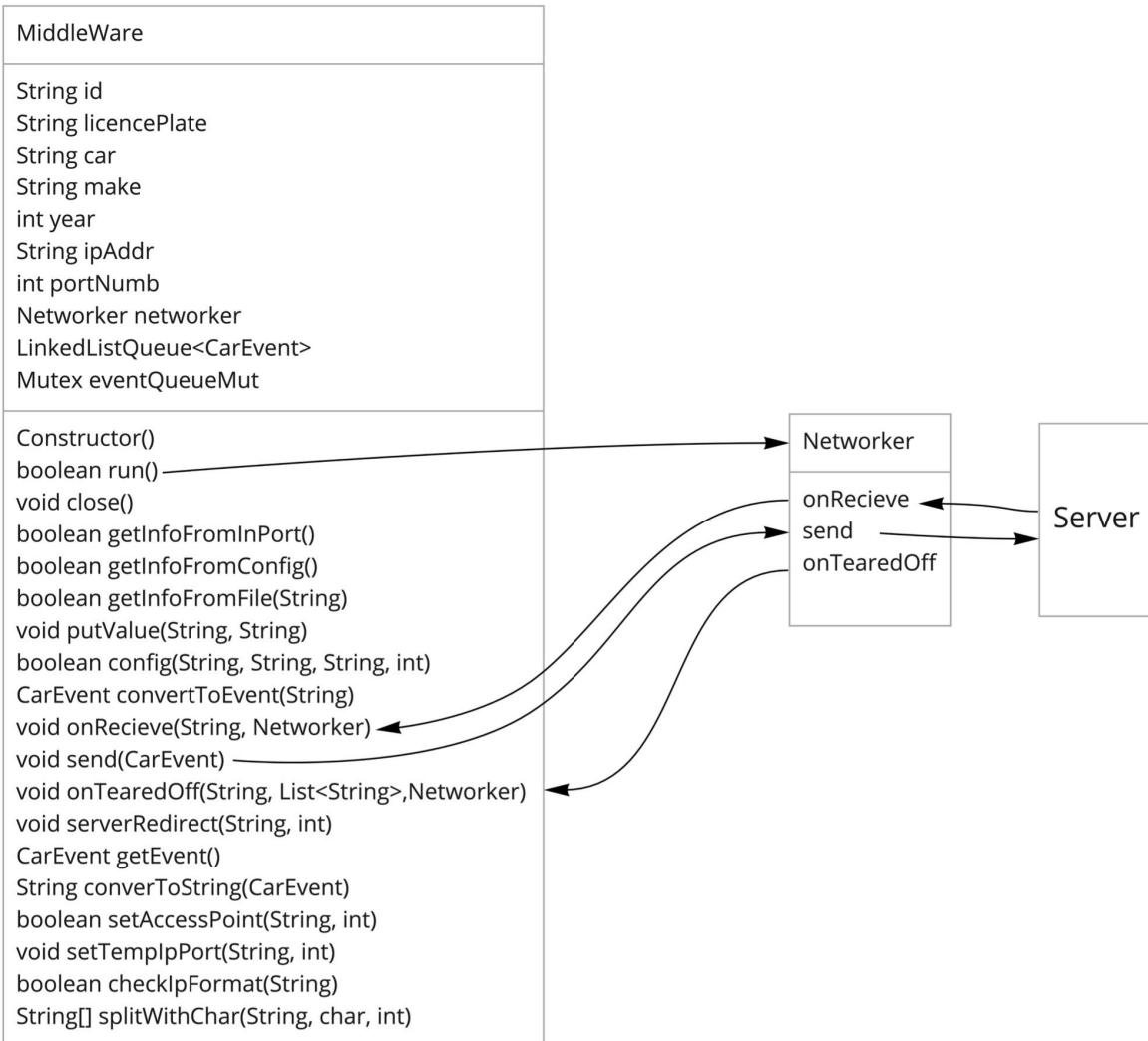
#### 5) OnRecieve

- 이 클래스는 `Networker` 메세지를 소켓의 `inputStream`을 사용하여 계속 읽습니다. 메세지를 읽으면, `Networker`의 `onReceive`함수를 호출하여 `Networker`에게 받은 메세지를 넘깁니다. 이 클래스도 `Thread class`를 implement 했습니다.

#### 6) NetworkerUser

- 이 인터페이스는 `Networker`를 사용하고 싶으면 이 interface를 implement한 클래스를 만들어야합니다. 이 interface에는 `onRecieve(String msg, Networker networker)` 와 `onTearedOff(String networkerId, List<String> unsentEvents, Networker networker)`가 있습니다.

ClientSide 클래스



### 1) MiddleWare

- 이 객체는 NetworkerUser 인터페이스를 implement했고, carEvent 객체를 사용하여 클라이언트와 직접소통하는 클래스입니다. Client가 서버에게 정보를 보내고 싶으면, CarEvent를 생성후에 MiddleWare의 send(CarEvent) 함수를 호출하면 됩니다. Client가 받은 Event를 보고싶으면, MiddleWare의 getEvent() 함수를 호출하면됩니다. 만약 아무 event도 안 받았다면, null을 return 합니다.
- 중요 파일:
  - config.txt 파일 - 이 파일은 자동차 정보를 저장하고 있습니다 ( licensePlate, car, make, year). MiddleWare 객체가 생성될때, 이 파일에서 정보를 읽고, field들에 값을 지정합니다. 이 파일이 없다면 생성이 됩니다. 만약 다른 종류의 정보를 변경하고 싶을시 MiddleWare 코드의 주석부분을 읽고, 바꾸시면 됩니다. 자동차의 정보를 변경하고 싶으시다면 미들웨어의 config(String licensePlate, String car, String make, int year) 함수를 호출하면 됩니다. 이 함수는 config.txt 파일의 내용을 변경합니다. 아니면 직접 그 txt 파일을 변경하셔도 됩니다

- ipPort.txt파일 - 이 파일은 ip주소와 포트 넘버를 저장하는 파일입니다. 이 파일의 내용을 변경하고 싶으면 MiddleWare의 setAccessPoint(String ip, int portNumb)를 호출하면 됩니다. 직접 txt 파일을 변경하셔도 상관없습니다.

두 위의 파일은 이러한 포맷을 씁니다 -> “데이터이름:데이터값\n”  
중간에 space를 넣지 말아주세요. ipPort.txt 와 config.txt를 참조해주세요.

#### Fields:

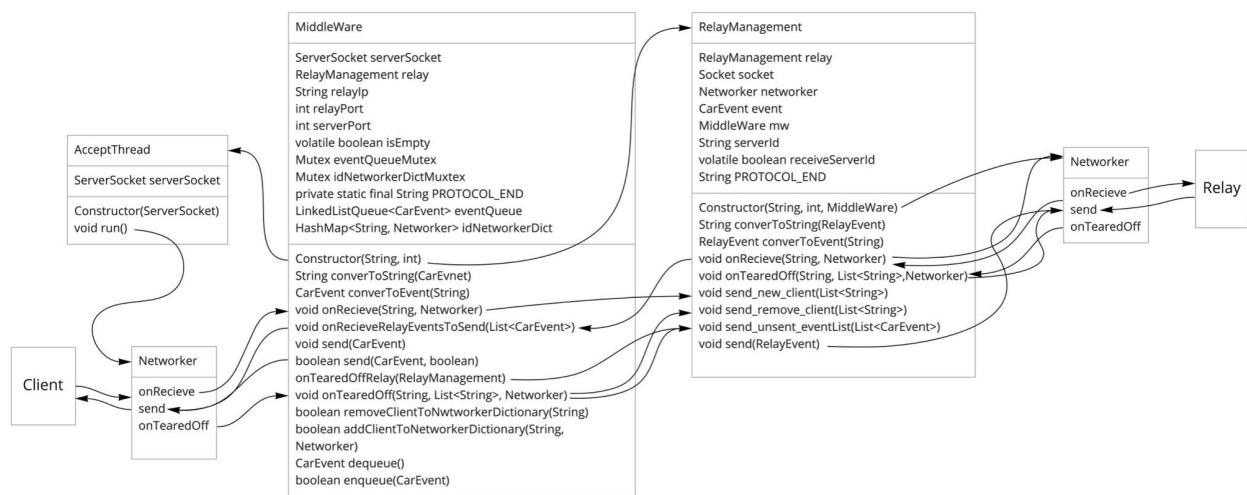
- String id - client의 id를 보관합니다. id를 만드는 방법은 미들웨어가 config.txt 파일안에 있는 자동차 정보를 읽고, id를 만듭니다. 자동차 정보는 config.txt에 저장이 됩니다. 만약 아이디를 생성하는 방법을 바꾸고 싶거나, MiddleWare에서 주석처리한 부분을 읽고 바꾸시면 됩니다. 현재 id를 생성하는 방법은 licensePlate.toUpperCase() + car.toUpperCase() + make.toUpperCase() + year 입니다. 지금은 클라이언트가 id가 없을 경우에는 서버에서 새로운 id를 받을 방법이 없습니다. 이 부분은 제가 나중에 서버용 미드웨어랑 클라이언트용 미들웨어를 바꿔서 가능하게 하도록 만들겠습니다.
- String licencePlate, String car, String make, int year - 이 변수들의 값은 config.txt에 저장이 되어있습니다. 만약 다른 자동차외의 다른 용도로 사용하고 싶으시면, 이 변수를 삭제하고 주석들을 읽어서 변경하시면 됩니다.
- ipAddr, portNum - ip주소와 포트넘버를 ipPort.txt에서 읽어옵니다. 만약, 처음으로 사용하시는 거라면, setAccessPoint(String ip, int portNumb)를 사용해서 ipPort.txt에 ip주소와 포트넘버를 써주세요. 만약 ip주소 포맷이 맞지 않는다면 에러가 뜹니다. (ipv4사용!)
- Networker networker - 사용할 네트워커 객체의 주소를 저장합니다.
- LinkedListQueue<CarEvent> eventQueue - 받은 메세지를 CarEvent객체로 변환해 저장합니다. 미들웨어의 getEvent()함수를 호출하면, 여기서 하나의 메세지를 dequeue합니다.
- Mutex eventQueueMut - 위의 객체는 여러개의 쓰레드가 사용하기 때문에, Mutex를 사용합니다.

#### methods:

- constructor() - config.txt 파일과 ipPort.txt 의 내용을 읽어서 위의 field들의 값을 지정합니다.
- config(String licensePlate, String car, String make, int year) - config.txt 파일의 내용을 바꿉니다.
- run() - 네트워커 객체를 생성하고, send, onReceive 쓰레드가 run 합니다.
- getEvent() 함수는 eventQueue에 있는 Event를 dequeue합니다. 이건 클라이언트 애플리케이션이 사용할 함수입니다.
- send(Event)는 클라이언트가 traffic을 서버한테 보낼때 사용합니다. 클라이언트와 미들웨어는 event를 사용하여 서로 통신을 합니다.
- convertToEvent(String)은 networker가 받은 메시지를 Event 객체로 변환합니다.
- converToString(Event)는 client가 middleWare한테 넘겨준 event객체를 String으로 변환합니다.
- onReceive(String, Networker)는 Networker가 메세지를 받을시 자기자신을 생성한 middleWare객체의 이 함수를 불러 middleware객체에게 받은 메시지를 넘깁니다. 이 함수는 converToEvent(String)을 호출후 받은 event를 eventQueue에다가 queue합니다. 그리고 client가 미드웨어의 getEvent()함수를 호출하면 그 queue에서 하나의 event를 dequeue합니다. 이 함수는 networker가 사용하는 callback 함수입니다.

- `onTearedOff(String id, List<String> unsentEvents, Networker networker)` - 이 함수도 위의 함수처럼 callback 함수입니다. Networker가 연결이 끊어졌을 때 사용하는 함수이고, id는 Networker가 사용하는 id입니다. 클라이언트 사이드 미들웨어에서는 사용하지 않습니다. unsentEvent는 네트워커가 보내지 못한 event들입니다. 현재는 보내지 못한 이벤트를 어떻게 처리할지는 정하지 못 했습니다.

## Server Side Middleware



(마지막 페이지에 더 크게 볼 수 있음)

## MiddleWare 클래스

- MiddleWare는 Client와 연결된 Networker와 MiddleWare에서 생성한 RelayManagement와 소통합니다. Client Networker로부터 받은 메시지로 CarEvent를 생성합니다. MiddleWare가 CarEvent를 처리할 수 있으면 처리하고, 처리할 수 없으면 RelayManagement로 넘깁니다.

### field

- ServerSocket serverSocket - 이 서버의 port에 대한 ServerSocket 값을 저장합니다.
- RelayManagement relay - 생성된 relay의 레퍼런스를 보관합니다.
- String relayIp - relay 측 IP 주소를 보관합니다.
- int relayPort - relay 측 port 번호를 보관합니다.
- int serverPort - 이 서버의 주소의 포트를 보관합니다.
- volatile boolean isEmpty - boolean의 volatile 형태입니다.
- Mutex eventQueueMutex - eventQueue에 대한 뮤텍스입니다.
- Mutex idNetworkerDictMutex - idNetworkerDict에 대한 뮤텍스입니다.

- private static vfinal String PROTOCOL\_END - 프로토콜의 끝을 명시합니다.  
여기서는 “\r\n”으로 표현됩니다.
- LinkedListQueue<CarEvent> evnetQueue;
- HashMap<String, Networker> idNetworkerDict - ID값과 Networker를 HashMap 형태로 저장합니다. 이 때, key 값이 ID, value가 Networker입니다.

## Method

- constructor(String, int) - RelayManagement를 생성 후, 자신의 주소를 이용해 소켓을 생성하며 AcceptThread를 생성해 클라이언트와 연결합니다.
- String converToString(CarEvent) - CarEvent를 String으로 변환하는 함수입니다.  
이때 PROTOCOL\_END를 덧붙여 String값을 반환합니다.
- CarEvent converToEvent(String msg) - String값을 CarEvent로 변환하는 함수입니다.  
만약 HEADER와 BODY가 비어있다면 null를 리턴하고, 아니면 CarEvent를 반환합니다.
- void onRecieve(String, Networker) - Networker측에서 호출하는 함수입니다. 이 함수는 Networker를 통해 받은 String값을 MiddleWare에서 관리합니다. 우선 Networker를 통해 받은 String값을 converToEvent를 통해 CareEvent로 변환합니다.  
이 CareEvent의 ID값을 추출한 후, 만약 ID값이 idNetworkerDict에 없다면 RelayManagement의 send\_new\_client를 호출해 ID값을 보낸 후,  
addClientToNetworkerDictionary를 통해 ID값을 idNetworkerDict에 추가합니다.  
RelayManagement에게 ID값을 보낼 때 idList를 생성해서 List형태로 보내게 됩니다.  
ID값이 기존에 있거나 idNetworkerDict에 추가한 후에는 enqueue를 통해 CarEvent를 eventQueue에 추가합니다.
- void onRecieveRelayEventsToSend(List<CarEvent>) - RelayManagement로부터 받은 CarEvent들을 send하는 함수입니다.
- void send(CarEvent) - send method를 호출하는 함수입니다.
- boolean send(CarEvent, boolean) - CarEvent를 Networker로 보내는 함수입니다.  
우선 CarEvent에서 ID값을 추출합니다. 만약 ID값이 idNetworkerDict에 있고, 그 ID값을 가지는 Networker가 닫히지 않았다면, CarEvent를 converToString을 통해 String으로 변환 후, networker의 send함수를 호출해 변환된 String값을 보낸 후 true를 리턴합니다. 만약 ID값이 idNetworkerDict에 있지만 Networker가 닫혀있다면 removeClientToNetworkerDictionary를 통해 ID값을 idNetworkerDict에서 삭제합니다.  
만약 ID값이 idNetworkerDict에 없거나, Networker가 close돼 ID값이 inNetworkerDict에 삭제되었다면, 그 ID값에 해당하는 CarEvent를 CarEvent List형태로 변환한 후, Relay의 send\_unsent\_eventList를 호출해 Relay로 보냅니다.  
Relay로 보내는데 성공하면 false를 반환합니다.
- void onTearedOff(String, List<String>, Networker) - Client와 연결된 Networker가 닫을 때 호출하는 함수입니다. 우선 removeClientToNetworkerDictionary를 호출해 ID값을 idNetworkerDict에서 제거합니다. 그 후, 제거된 ID를 RelayManagement의

`send_remove_client`를 호출해 Relay에도 이 ID값을 제거합니다. 그 ID가 못 보낸 CarEvent는 relay측의 `send_unsent_eventList`를 통해 Relay로 보냅니다.

- boolean `removeClientToNetworkerDictionary(String)` - String으로 받은 ID값을 idNetworkerDict에 삭제하는 함수입니다. 이 때, idNetworkerDictMutex를 사용해 race Condition을 피했으며, 삭제하는데 성공하면 true, 실패하면 false를 리턴합니다.
- boolean `addClientToNetworkerDictionary(String, Networker)` - ID값과 그 ID를 가지는 Networker를 idNetworkerDict에 추가하는 함수입니다. 이 때, idNetworkerDictMutex를 사용해 race Condition을 피했으며, 추가하는데 성공하면 true, 실패하면 false를 리턴합니다.
- CarEvent `dequeue()` - eventQueue에 있는 CarEvent를 제거한 후, 제거된 CarEvent를 리턴하는 함수입니다. 이 때, eventQueueMutex를 사용해 race Condition을 피했으며, 추가하는데 성공하면 삭제된 CarEvent를 리턴합니다.
- boolean `enqueue(CarEvent)` - CarEvent를 eventQueue에 추가하는 함수입니다. 이 때, eventQueueMutex를 사용해 race Condition을 피했으며, 추가하는데 성공하면 true, 실패하면 false를 리턴합니다.

#### AcceptThread class 클래스

- MiddleWare에서 생성하는 객체이며, Client의 연결을 기다립니다.. Thread를 이용해 끊김없이 Client의 호출을 기다릴 수 있습니다.

##### field

- Socket socket - accept()를 통해 연결된 Client의 socket값을 저장합니다.
- ServerSocket serverSocket - MiddleWare의 serverSocket값을 저장합니다.
- NetworkerUser mw - MiddleWare의 reference를 저장합니다.
- constructor(ServerSocket, MiddleWare) - 매개변수로 입력받은 ServerSocket을 이 클래스의 ServerSocket에, MiddleWare를 NetworkerUser에 저장합니다.
- void run() - Client의 연결을 기다리는 함수입니다. 만약 serverSocket이 새로운 Client의 호출을 받으면, 그 Client에 대한 Networker를 생성하고, Networker는 MiddleWare와 통신하게 됩니다.

#### RelayManagement 클래스

- RelayManagement는 MiddleWare와 Relay랑 연결된 Networker와 통신하는 객체입니다. MiddleWare에서 처리하지 못하는 서비스는 RelayManagement를 통해 RelayEvent를 생성한 후, Relay와 연결된 Networker를 통해 Relay 서버와 통신합니다. 반대로 MiddleWare는 그 Networker와 연결된 Relay로부터 RelayEvent를 받아 CarEvent로 변환 후 MiddleWare에게 보냅니다.

##### field

- RelayManagement relay - RelayManagement의 레퍼런스를 저장합니다.
- Socket socket - soket값을 저장합니다.
- Networker networker - Networker 레퍼런스를 저장합니다.
- CarEvent event - CarEvent를 저장합니다.

- MiddleWare mw - 이 RelayManagement를 생성한 MiddleWare의 레퍼런스 값을 저장합니다.
- String serverId - 이 서버의 Id 값을 저장하는 필드입니다.
- volatile boolean receiveServerId - ServerId값을 받았는지 확인하기 위한 volatile boolean 변수입니다.
- private static final String PROTOCOL\_END - 프로토콜의 끝을 명시합니다. 여기서는 “\r\n”으로 표현됩니다.

#### Method

- constructor(String, int, MiddleWare) - relayIp와 relayPort를 이용해 Socket을 생성 후, Relay와 통신하는 Networker를 생성합니다. 이 때, Relay에게 serverId 값을 받을 때까지 다음 코드를 실행하지 않습니다.
- convertToString(RelayEvent) - RelayEvent를 String으로 변환하는 함수입니다. MiddleWare 측의 convertToString을 이용합니다.
- RelayEvent convertEvent(String) - String값을 RelayEvent로 변환하는 함수입니다. 만약 HEADER와 BODY가 비어있다면 null을 리턴하고, 아니면 생성된 RelayEvent를 반환합니다.
- void onReceive(String, Networker) - Relay 측 Networker에게 받은 String값을 convertEvent를 통해 RelayEvent를 생성합니다. 만약 RelayEvent 값의 null이라면 그냥 리턴합니다. RelayEvent가 null이 아니면, 생성된 RelayEvent의 reqNo를 추출합니다. reqNo가 0이면, Relay로부터 serverId 값을 받아 이 객체의 serverId에 저장합니다. 그 후, receiveServerId를 true로 변경해 constructor의 while문을 벗어나게 합니다. 만약 reqNo가 1이라면, MiddleWare를 통해 Client에게 보내야 할 CarEvent들을 Relay에게 받습니다. Relay는 CarEvent를 리스트 형태로 보내며, 만약 리스트가 비어있으면 그냥 리턴하고 아니면 MiddleWare의 onReceiveRelayEventsToSend를 호출해 CarEvent List를 MiddleWare에게 보냅니다. 만약 reqNo가 2라면, 다른 Relay를 연결하라는 의미입니다. Relay부터 새로 연결할 Relay의 IP와 Port 번호를 받은 후, 현재 Relay와 소통하는 Networker를 닫고, 새로운 Networker를 생성합니다.
- void send\_new\_client(List<String>) - Relay에게 새로운 Client Id값을 보내는 함수입니다. 새로운 RelayEvent를 생성 후, RelayEvent의 putClientId를 통해 Client Id List를 입력하고, ReqNo를 0으로 설정하고, 현재 객체의 serverId를 RelayEvent의 serverId 값으로 설정한 후, send를 호출해 Relay에게 RelayEvent를 보냅니다.
- void send\_remove\_client(List<String>) - Relay에게 제거된 Client Id값을 보내는 함수입니다. 새로운 RelayEvent를 생성 후, RelayEvent의 putClientId를 통해 Client Id List를 입력하고, ReqNo를 1로 설정하고, 현재 객체의 serverId를 RelayEvent의 serverId 값으로 설정한 후, send를 호출해 Relay에게 RelayEvent를 보냅니다.
- void send\_unsent\_eventList(List<CarEvent>) - Relay에게 보내지 못한 CarEvent들을 보내는 함수입니다. 새로운 RelayEvent를 생성 후, RelayEvent의 putCareEvents를 통해 CarEvent List를 보내고, ReqNo를 2로 설정하고, 현재 객체의

serverId를 RelayEvent의 serverId 값으로 설정한 후, send를 호출해 Relay에게 RelayEvent를 보냅니다.

- void send(RelayEvent) - RelayEvent를 cnoverToString을 통해 String으로 변환한 후, Networker의 send를 통해 Relay에게 보냅니다.

## RelayEvent 클래스

- 이 객체는 RelayManagement와 Relay가 서로 주고받는 메시지 역할을 하는 객체입니다. CarEvent를 확장한 것으로써, HEADER에는 ServerID가 있으며, BODY에는 CLIENTID, REQNO, IPADDR, PORTNUMB 등이 들어갈 수 있습니다. RelayManagement와 Relay는 RelayEvent의 REQNO를 통해서 어떤 작업을 해야하는지 알 수 있습니다.

### field

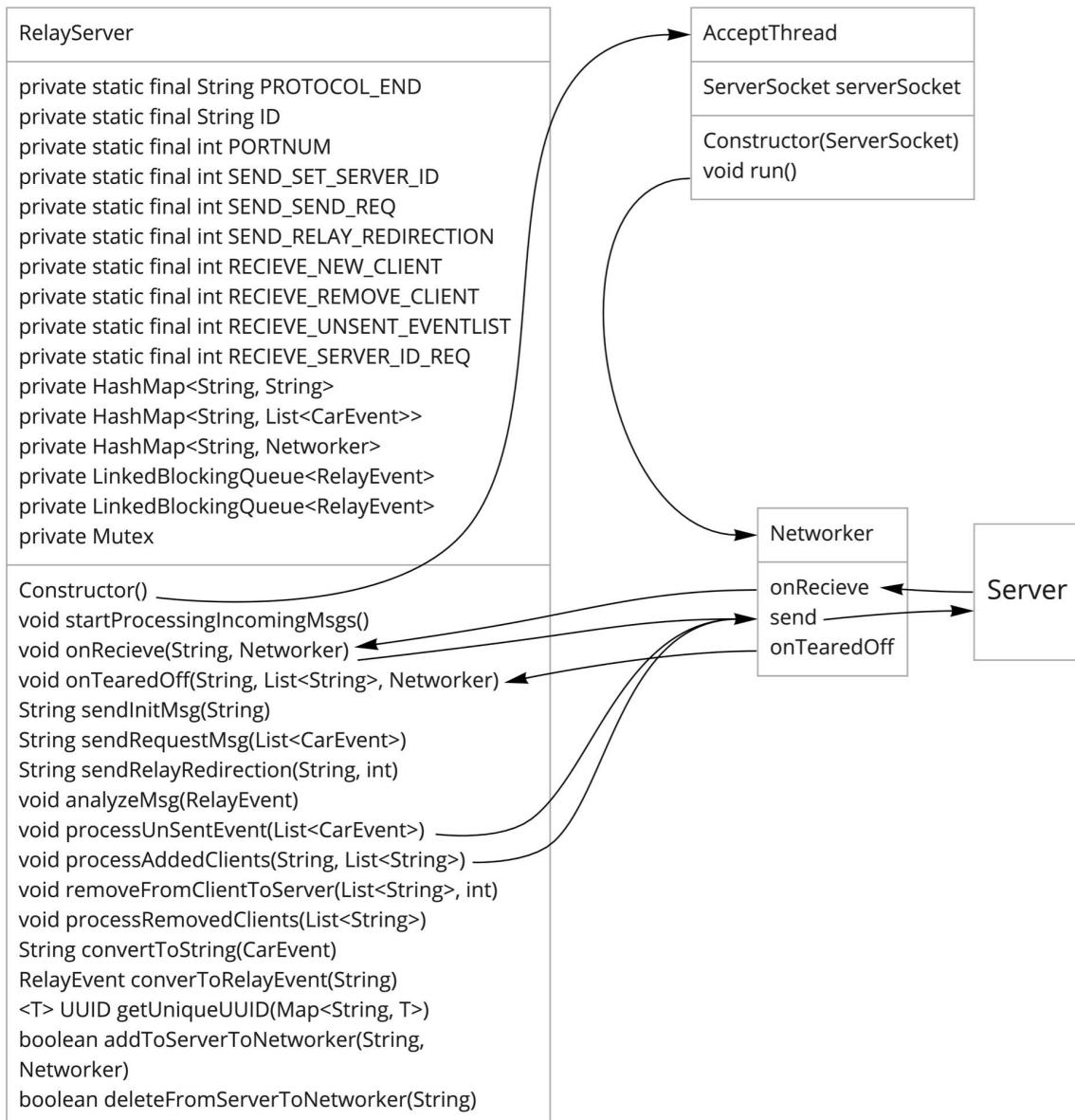
- protected static final String EVENTLIST = "EVENTLIST"
- protected static final String CLIENTID = "CLIENTID"
- protected static final String REQNO = "REQNO"
- protected static final String IPADDR = "IPADDR"
- protected static final String SERVERID = "SERVERID"
- protected static final String PORTNUM = "PORTNUMB"

### Method

- RelayEvent() - CarEvent의 constructor()를 호출합니다.
- RelayEvent(JSONObject) - CaerEvent의 constructor(JSONObject)를 호출합니다.
- putCareEvents<List<CarEvent>()) - RelayEvent의 BODY에 있는 EVENTLIST에 CarEvent List를 추가하는 함수입니다. 성공하면 true를, 실패하면 false를 리턴합니다.
- LinkedList<CarEvent> getCareEvents() - RelayEvent의 BODY에 있는 EVENTLIST의 값을 CarEvent List로 변환 후 리턴하는 함수입니다. 변환에 성공하면 LinkedList<CarEvent>로 반환합니다.
- boolean putClientId(List<String>) - RelayEvent의 BODY에 있는 CLIENTID에 Client Id List 값을 추가하는 함수입니다. 성공하면 true를 반환하고 실패하면 false를 반환합니다.
- LinkedList<String> getClientId() - RelayEvent BODY의 CLIENTID값을 LinkedList<String> 으로 변환 후 그 값을 리턴하는 함수입니다.
- boolean putReqNo(int) - RelayEvent header의 REQNO에 값을 집어넣는 함수입니다. 성공하면 true를, 실패하면 false를 리턴합니다.

- `getReqNo()` - RelayEvent header의 REQNO값을 알아내는 함수입니다. header의 REQNO 값을 리턴합니다.
- `boolean putServerId(String)` - RelayEvent의 HEADER에 SERVERID값을 설정하는 함수입니다. 성공하면 `true`를, 실패하면 `false`를 리턴합니다.
- `String getServerId()` - RelayEvent의 HEADER에 있는 SERVERID값을 리턴하는 함수입니다. 성공하면 SERVERID를, 실패하면 `null`을 리턴합니다.
- `boolean putIpAddr(String)` - RelayEvent의 BODY에 IPADDR 값을 설정하는 함수입니다. 성공하면 `true`를, 실패하면 `false`를 리턴합니다.
- `String getIpAddr()` - RelayEvent의 BODY에 있는 IPADDR값을 리턴하는 함수입니다. 성공하면 IPADDR값을, 실패하면 `null`을 리턴합니다.
- `boolean putPortNum(int)` - RelayEvent의 BODY에 PORTNUM 값을 설정하는 함수입니다. 성공하면 `true`를, 실패하면 `false`를 리턴합니다.
- `int getPortNum()` - RelayEvent의 BODY에 있는 PORTNUM 값을 리턴하는 함수입니다. 성공하면 PORTNUM값을, 실패하면 -1을 리턴합니다.

## RelayServer 클래스



- Relay 서버는 serverSide에서 연결되지 않은 client에게 보내야 할 event들을 보관하고, 다른 서버에서 해당 client에 연결이 되면, 그 서버에게 event를 보내, 서버가 해당 클라이언트에게 메세지를 보내게 해주는 서버입니다. 이 서버는 클라이언트와 연결하는 서버들과 연결을 합니다. 이 서버가 필요한 이유는 여러개의 서버를 사용할 때, 한 서버가 다른 서버와 연결돼 있는 클라이언트에게 event를 보내고 싶을 때나 아직 서버와 연결이 되지 않는 클라이언트에게 메시지를 보내고 싶을 때 필요합니다. 서버쪽 미들웨어가 메세지를 보내야 할 클라이언트와 연결이 되어 있지 않으면 Relay 서버에게 보냅니다.

Field:

- **private final field:**
  - 릴레이 서버가 서버쪽에게 보낼 Request number
  - `SEND_SET_SERVER_ID` - 릴레이 서버가 서버에게 서버 아이디를 부여할 때
  - `SEND_SEND_REQ` - 릴레이 서버가 서버에게 CarEvent들을 보내달라고 할 때
  - `SEND_RELAY_REDIRECTION` - 서버에게 새로운 릴레이 서버랑 연결하라고 할 때

서버가 릴레이 서버에게 보낼 Request Number

RECEIVE\_NEW\_CLIENT - 릴레이 서버에게 새로운 클라이언트랑 연결을 맺었다고 알림

RECEIVE\_REMOVE\_CLIENT - 릴레이 서버에게 해당 클라이언트랑 연결이 끊겼다고 말함

RECEIVE\_UNSENT\_EVENTLIST - 릴레이 서버에게 보내지 못한 CarEvent를 보냄

RECEIVE\_SERVER\_ID\_REQ - 릴레이 서버에게 서버 아이디를 보내달라고 요청

릴레이 서버가 사용할 데이터 구조:

- HashMap<String, String> clientToServer - 키는 ClientId, value는 ServerId
- HashMap<String, List<CarEvent>> clientToEventList - 키는 clientId, value는 List<CarEvent>, 이건 서버들이 못보낸 CarEvent를 clientId로 루어서 저장
- HashMap<String, Networker> serverToNetworker - serverId를 해당 서버랑 통신하는데 사용하는 Networker랑 같이 저장
- LinkedBlockingQueue<RelayEvent> msgQueue - Networker가 서버에게 Event를 받으면 이 BlockingQueue에 저장합니다.
- LinkedBlockingQueue<RelayEvent> unsentMsgQueue- Networker가 서버에게 보내야 할 RelayEvent를 보내지 못할 때 여기에 저장합니다. 현재는 저장만하고 이 메세지를 processing하지 않습니다.
- Mutex mutexForServerToNetworker - serverToNetworker 데이터 구조를 업데이트 할 때 사용하는 Mutex이다. 여러 쓰레드에서 사용하기 때문에 필요
- AcceptThread (inner class) - 이건 이너클래스이다. 이 쓰레드 객체에게 ServerSocket을 넘기고, 이 쓰레드는 ServerSocket을 사용해서 TCP connection을 받는다. 새로운 TCP connection을 받으면 Networker 객체를 생성합니다.

Methods:

- Constructor() - 위의 데이터 구조들을 initialize하고, AcceptThread 객체를 생성하여 계속 받습니다.
- startProcessingIncomingMsgs() - 이것은 blocking method이다. 이 함수를 호출하면, msgQueue에서 메세지를 retrieve하고, 메세지가 있으면 프로세싱을 합니다. 이 함수는 analyzeMsg() 함수를 호출해서 받은 메세지를 프로세싱합니다.
- analyzeMsg(RelayEvent) - 이 함수는 RelayEvent의 ReqNo를 읽고, ReqNo에 맞는 함수를 호출합니다. 그 함수들은 아래 세 개 함수들입니다.
- processUnsentEvent(List<CarEvent> carEvent) - 이건 서버가 보내지 못한 CarEvent들을 프로세싱하는 것입니다. 각 CarEvent들의 id를 보고, clientToServer 데이터 구조를 확인하여, 해당 클라이언트랑 연결되어 있는 서버가 있으면, 그 서버에게 CarEvent들을 보냅니다. 아니면 clientToEventList에 저장합니다.
- processAddedClients(String serverId, List<String> clientIds) - 서버가 새로운 클라이언트랑 연결될 때 이 함수를 호출합니다. clientToServer에 각 clientId와 serverId를 pair로 해서 저장을 하고, 만약 clientIds 리스트 안에 있는 clientId가 clientToEventList에 있다면, 그 클라이언트에게 보내야 할 CarEvent들을 그 서버에게 보냅니다.
- processRemovedClients(List<String> clientIds) - 서버가 클라이언트랑 연결이 끊겼을 때, 서버가 끊긴 클라이언트 아이디들을 보냅니다. clientToServer에서 clientId가 key인 entries들을 다 지웁니다.
- onReceive(String msg, Networker networker) - Networker가 사용할 callback 함수입니다. 받은 msg를 RelayEvent로 변환하고, eventQueue에 넣습니다.

- `onTearedOff(String serverId, List<String> unsentMsgs, Networker networker)` - 못보낸 event들을 RelayEvent로 변환하고, `unsentQueue`에 넣습니다. 그리고 해당 `serverId`를 key로 갖는 `serverToNetworker`의 entry를 지웁니다.
- `sendInitMsg(String serverId)` - 서버에게 argument로 받은 ID를 보낼 메세지를 만듭니다.
- `sendRequestMsg(List<CarEvent> carEvent)` - 해당 서버가 보낼 CarEvent들을 해당 서버에게 보낼 메세지를 만듭니다.
- `sendRelayRedirection(String ipAddr, int portNum)` - 이건 지금 사용하고 있지 않지만, 나중에 여러개의 RelayServer를 사용할것을 대비하여, 만든 함수입니다. 이 함수는 서버가 새로 연결할 릴레이서버의 IP주소와 포트넘버를 메세지에 넣고 String을 변환합니다.

## B. Protocols Used

**UTF-8 encoding is used!**

```

- Client -> Server protocol
{
  HEADER:
  {
    ID: __,
    EVENTNO: __,
    PRIORITYNUMBER: __
  }
  BODY:
  {
    COORDINATE: __
  }
}
/r/n

```

- 이 client에서 서버로 보낼때 필수로 있어야 되는 field는 Id field와 Header, Body가 포함이 되어야 합니다. 나머지는 다 optional입니다. EventNo는 어떤 event를 알리는지 표시하기 위해서이고, PriorityNumber는 그 event가 얼마나 급한지 표시하기 위해서입니다. Coordinate는 그 자동차의 좌표를 나타냅니다. 하나의 event의 시작은 '{', 끝은 '}'로 표시가 됩니다. 만약 다수의 event들을 하나의 message로 보내고 싶으면 event 사이에 '\n,,\n'을 넣으면 됩니다. 그리고 메세지의 끝을 알리려면 '/r/n'을 넣으면 됩니다.

- server -> client protocol

```

{
  HEADER:
  {
    ID: _____,
    PRIORITYNUMBER: _____,
    COMPANYNAME: _____,
    TIMESTAMP: _____,
    EVENTNO: _____,
  }
  BODY:

```

```
{  
}  
}  
\r\n
```

- server에서 client로 보내는 protocol도 비슷합니다. 다른점은 어떠한 field도 필수로 넣어야되지 않습니다. 여기서 TimeStamp는 그 Event가 일어난 시간을 나타냅니다. Header와 Body 부분만있고 하나의 event는 '{'와 '}' 기호로 시작과 끝을 알리면됩니다.

Server -> Client protocol 특수 케이스 1:

```
{  
    HEADER:  
        {  
            ID:_____,  
            EVENTNO: 0,  
        }  
    BODY:  
        {  
            IPADDR:_____,  
            PORTNUM:_____,  
        }  
}
```

\r\n

- server가 EVENTNO가 0 일시에는 지금 연결하고있는 socket을 새로운 ip주소와 portnumber로 연결하라는 뜻이므로, Client side app에 메세지를 올려보내지 않고, MiddleWare가 ip 주소와 port number를 바꿉니다.

Server -> Client protocol 특수 케이스 2:

```
{  
    HEADER:  
        {  
            ID: 0,  
        }  
    BODY:  
        {  
        }  
}
```

\r\n

- ID가 0인경우에는 하나의 메세지를 모든 client로 보내라는 뜻입니다. 따라서 자기가 맡고있는 모든 client들과 relay에게 메세지를 보냅니다. 하지만, 지금은 현재 broadcast 서비스를 지원하지 않습니다. 이건 제가 추후에 만들도록 하겠습니다.

- server와 server끼리의 protocol
 

```
{
        HEADER:
        {
          SERVERID: _____
        }
        BODY:
        {
        }
      } \r\n
```
- server가 새로운 connection을 다른 서버와 맺을 때 서버끼리의 connection을 인식하게 하기 위해서 'ServerId' field가 있어야 됩니다.

여기서 EventNumber에 따라서 포함해되는 field가 바뀝니다.

Server에서 Client로 보낼 때, EventNo가 0일 시에는 Client MiddleWare는 ip주소와 portNumber를 바꾸고 새로운 TCP connection을 맺습니다. 받은 메시지는 client로 보내지지 않습니다. 그리고 Server가 EventNumber가 0인 메시지를 보내면, Body에는 'IpAddressIPv4'와 'PortNumber'라는 field를 꼭 포함시켜야 합니다. 이렇게 한 이유는 client에게 server가 바뀌는 것을 모르게 하려는 것입니다.

그리고 EventNumber 1~1000은 차량사고에 관련한 EventNumber로 지정을 하였습니다. 여기서 PriorityNumber는 -20(심각한 사고), -19(가벼운 사고)로 지정을 하였습니다. 이것은 꼭 지켜야 할 것은 아닙니다.

Relay -> Server protocol

Request Number summary:

For Relay -> Server msg

```
private static final int SEND_SET_SERVER_ID = 0;
private static final int SEND_SEND_REQ = 1;
private static final int SEND_RELAY_REDIRECTION = 2;
```

For Server -> Relay msg

```
private static final int RECIEVE_NEW_CLIENT = 0;
private static final int RECIEVE_REMOVE_CLIENT = 1;
private static final int RECIEVE_UNSENT_EVENTLIST = 2;
private static final int RECIEVE_SERVER_ID_REQ = 3;
```

Relay -> Server

```
{
  HEADER:
  {
    REQNO: //0, 1, or 2이 될 수 있다
    SERVERID: //REQNO 0일 때만 보냄(서버에게 ID부여)
  }
  BODY:
  {
```

```

EVENTLIST: [ , , ] //REQNO가 1일때만 보냄. 서버에게 해당
//CarEvent들을 보내라고 요청

IPADDR: //REQNO가 2일때
PORTNUMB: //REQNO가 2일때. 새로운 Relay서버와 연결을
//맺으라고 요청
}

}\r\n

```

- 0 -> 서버에게 ID를 지정  
 1-> 서버에게 CarEvent들을 보내라고 요청  
 2-> 서버에게 새로운 RelayServer와 연결을 맺으라고 요청

```

Server -> Relay
{
  HEADER:
  {
    REQNO: //0, 1, or 2,3이 될수있다
    SERVERID: //항상 자기 서버 아이디를 보냄
  }
  BODY:
  {
    EVENTLIST: [ , , ] //REQNO가 2일때만 보냄. 릴레이서버에게
    //못보낸 CarEvent들을 보냄
    CLIENTID:[ , , ] // REQNO가 0과 1일때 보냄
    //릴레이서버에게 새로운 클라이언트나
    //연결이 끊긴 클라이언트를 알림
  }
}\r\n

```

- 0 -> 서버가 새로운 클라이언트랑 연결될때 릴레이서버에게 보냄  
 1-> 서버가 연결이 끊긴 클라이언트를 릴레이서버에게 보냄  
 2-> 서버가 보내지 못한 CarEvent들을 릴레이서버에게 보냄  
 3-> 서버가 ID를 받지못하면 보낸다. 이때는 REQNO만 3으로 바꾸기만 하면된다.

### \*\*\* 추가해야될점:

- server나 클라이언트가 메세지를 받으면 Response 메세지를 할수있게끔 바꾼다.
- server가 일시적인 오류로 RelayServer와 연결이 끊기고 다시 연결을 맺으면, Server가 현재 가지고있는 모든 clientId들을 보낼수 있게 한다
- broadcast 서비스 구현
- 여러개의 RelayServer를 사용할수있게끔 바꾸기
- RelayServer가 fail할수도 있으니, 못보낸 Event들을 Database에 저장하고, RelayServer는 caching할수 있게끔 만든다.

- 클라이언트에서 서버에게 새로운 id를 받고싶을시 server와 client 미들웨어에 코드를 바꿔서 구현한다 -> 일부분은 Server측 미들웨어를 사용할때, 특정한 interface를 implement한 객체를 Server측 미들웨어에 넘기도록 한다. 이 인터페이스는 미들웨어가 호출할수 있는 callback함수가 있어야한다. (예를들어 String onGenerateNewId()). 서버측에서 새로운 id를 생성하는 방법을 정할수 있다.
- 서버나 클라이언트가 메세지를 받을때 blockingQueue를 써서 더 효과적이게 cpu를 사용하게 만들기!
- CarEvent에 JSONArray나 JsonObject, float, int 타입의 데이터를 넣게 하기. 추가로 putEventNo나 putId 함수도 구현하기

#### 4) 사용방법

- ClientSide:

ex)

```

CarEvent temp1;
MiddleWare temp = new MiddleWare();
if(!temp.run())
{
    System.err.println("Failed to connect to the server");
}

System.out.println("start sending!");
CarEvent eve = new CarEvent();
eve.putHeader("ID", "ClientEclipse");

temp.send(eve);

while(true)
{
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e)
    {
        System.err.println("failed to sleep");
        e.printStackTrace();
    }

    temp1 = temp.getEvent();
    if( temp1 != null)
    {
        System.out.println(temp1);
    }
}

```

1) 미들웨어 객체를 생성한다.

2) MiddleWare.run() 함수를 호출 -> True를 리턴하면 실행성공, 아니면 실행실패

- 3) CarEvent 객체를 사용해서 자기 자신의 id를 넣고, 나머지 원하는 데이터를 key-pair 형식으로 넣는다. 현재는 String값들만 넣을수 있게했다.
  - 4) MiddleWare.send(CarEvent)함수를 호출한다. 그러면 해당 CarEvent를 서버에게 보내는 것이다.
  - 5) 받은 메세지를 확인하고 싶으면 MiddleWare.getEvent()함수를 호출해야한다. 이 함수는 받은 메세지가 없으면 null을 리턴한다.
- ServerSide:
- ex)
- ```

MiddleWare mw = new MiddleWare("172.31.10.90", 6000);

CarEvent carEvent = new CarEvent();
carEvent.putHeader("ID", "ClientEc2");
carEvent.putHeader("SENDMESSG", "from server1");
mw.send(carEvent);

while(mw.isEmpty);
System.out.println("out of move");

CarEvent tempEve;
tempEve = mw.dequeue();
System.out.println("dequeued msg: " + tempEve);

```
- 1) 서버쪽 미들웨어 객체를 생성한다. -RelayServer의 ip주소와 포트넘버 필요
  - 2) CarEvent에 보내야될 클라이언트 id를 넣고, 원하는 데이터를 추가한다
  - 3) MiddleWare.send(CarEvent)를 사용하여 CarEvent를 보낸다
  - 4) MiddleWare에서 메세지를 받을때, MiddleWare.isEmpty field를 사용하여 미들웨어가 메세지를 받았는지 체크
  - 5) MiddleWare.dequeue() 함수를 호출하여 받은 CarEvent를 받는다.

- RelayServer:
- Ex)
- ```

RelayServer relServer = new RelayServer();
relServer.startProcessingIncomingMsgs();

1) 위 코드를 RelayServer 클래스의 Main 함수에 그대로 넣어서 사용해도 된다
2) RelayServer 객체를 생성후, RelayServer.startProcessingIncomingMsgs()를
호출하면 된다. 이 함수는 RelayServer를 실행시킨다.

```

