소켓 프로그래밍

목차

1. 소켓이란?
2. 소켓 사용 방법
3. 소켓 프로그래밍 실습
    1. 프로젝트 생성
    2. 소켓 서버
    3. 소켓 클라이언트
    4. 프로젝트 디버깅
4. 문제점
5. 해결방법

1. 소켓이란?

우리는 앞으로 소켓이라는 것 사용해서 통신할 것이다.
그럼 소켓이 뭘까?
소켓이 무엇인지 알기 전에 우리가 네트워크 통신을 하려면 신경써야 할 것들이 무엇인지 잠깐 알아보도록 하자.
우선 OSI 7계층을 바탕으로 전기 신호, 하드웨어, 운영체제 등 굉장히 많은 부분을 신경써야 한다. 이를 해결하기 위해 1971년 UNIX에서 개발한 프로그래밍 인터페이스가 소켓이다.

2. 소켓 사용 방법

그래서 소켓은 어떻게 쓰는 걸까?
앞서 예시로 들었던 전화에 비유해보자.
우린 수화기를 들어 원하는 목적지를 전화번호라는 주소를 사용해서 입력할 것이다.
그럼 우린 수화기만 들고 있어도 나의 수화기와 나의 수화기의 목적지가 되는 상대방의 수화기는 알아서 연결이 될 것이다.

그럼 이 예시를 소켓에 대입해보자.
전화를 하는 주체는 소프트웨어 즉 우리가 만든 게임과 같은 응용 프로그램이 될 것이다.
그리고 목적지의 주소를 입력하기 위한 전화번호는 IP 주소와 포트 번호의 조합으로 이루어진다.
응용 프로그램은 단지 주소를 입력하였을 뿐이지만 각 수화기 즉 소켓들은 알아서 연결이 될 것이다.
이처럼 편리하게 네트워크 통신을 할 수 있도록 해주는 것이 소켓이다.

3. 소켓 프로그래밍 실습

자 그럼 소켓을 사용하여 보낸 메세지를 그대로 돌려주는 에코 서버와 클라이언트를 만들어 보자.
상세한 코드는 피피티를 참고하길 바란다.

첫 서버 프로그래밍인 것을 감안하여 본 강의에선 모든 부분을 자세히 서술하겠지만, 앞으로의 강의에선 불필요한 부분은 제거하고 설명할 것이다.
이번 강의를 집중하여 수강해서 소켓 프로그래밍의 흐름을 이해하고 서버 프로그래밍에 대한 감을 잘 잡도록 하자.

1. 프로젝트 생성

Server라는 이름의 프로젝트를 SocketProgramming 솔루션에 생성한다.
닷넷 버전은 6버전을 사용할 것이고, 자세한 라이브러리 탐색을 위해 최상위 문은 사용하지 않도록 하자.

2. 소켓 서버

이번 프로젝트에선 다음과 같은 라이브러리를 사용한다.

1. System
2. System.Net
3. System.Net.Sockets
4. System.Text

우선 클라이언트들을 받아드릴 리스닝 소켓을 생성해야 한다.
현재 IP에 소켓을 생성하기 위해 현재 IP 주소를 받아온다.
찾아 온 현재 컴퓨터의 IP에서 8081포트로 백로그가 1인 소켓을 생성한다.

위와 같은 방법으로 생성한 소켓을 앞으로 listenSocket이라 부르도록 하자.
listenSocket으로 접속을 요청한 다른 소켓 즉 클라이언트를 수락한다.
통신하는 작업을 끝낸 서버는 꺼버려 차지하고 있던 메모리를 지워주도록 하자.

이러한 과정을 통해 서버의 메인 스레드는 다음과 같은 흐름을 갖게 된다.

1. listenSocket 생성
2. client 수락
3. client와 통신
4. listenSocket 종료 (서버 종료)

통신하는 과정으로 넘어가보자.
우선 주고 받을 메세지를 담을 저장공간이 필요하다.
네트워크 상에선 모든 데이터가 byte의 배열로 송수신 되기 때문에 byte의 배열로 저장공간, 즉 buffer를 생성한다.

clientSocket으로부터 온 데이터를 buffer에 담아 UTF8로 인코딩하여 문자열로 바꾼다.
서버는 받은 메세지를 출력하고, 받은 메세지에 'exit'이라는 문자열이 포함되어있지 않다면 메세지를 echo해야 한다.
만약 'exit'라는 문자열이 포함되어 있다면 통신을 종료할 수 있도록 false를 반환하자.
echo할 땐 서버가 보낸 메세지라고 식별할 수 있도록 하는 문자열을 추가해 다시 인코딩하자.
인코딩 되어 byte의 배열의 형태를 띄고 있는 메세지를 clientSocket에게 전송한다.

모든 과정이 순조롭게 진행되었다면 true, 중간에 문제가 생겨 catch문에 걸렸다면 서버를 종료할 수 있도록 false를 반환해주자.

다시 메인으로 돌아와 통신하는 부분에 방금 만든 함수를 포함시켜 우리가 설계했던 흐름대로 작동하도록 해두자.

3. 소켓 클라이언트

다음은 클라이언트를 만들 차례이다.
Server와 별개로 작동할 수 있도록 Client라는 이름의 프로젝트를 생성한다.
사용하는 라이브러리는 서버를 제작할 때와 동일하다.

이번에도 서버 때와 마찬가지로 소켓을 먼저 생성해야 한다.
하지만 서버 쪽에서 생성했던 listenSocket과는 조금 다른 형태로 생성한다.
백로그를 둔 상태로 리스닝 시켜 다른 소켓을 받아드리는 것이 아닌,
연결할 IP와 포트를 입력하여 특정 소켓에 접속할 수 있도록 하는 소켓을 생성한다.
하지만 서버 쪽에서 로컬 컴퓨터의 IP로 listenSocket을 생성했기에 해당 소켓도 같은 IP로 생성해주도록 하자.

메인으로 넘어와 생성한 소켓을 서버의 소켓의 주소를 종단점으로 두어 연결을 시도한다.

이러한 과정을 통해 클라이언트의 메인 스레드는 다음과 같은 흐름을 갖게 된다.

1. serverSocket 생성
2. serverSocket을 listenSocket과 연결
3. 서버와 통신
4. serverSocket 종료 (서버와 연결 끊기)

통신하는 부분을 만들어보자
서버의 통신하는 부분과 크게 다를 것 없다.
서버가 통신하는 과정은 메세지를 받고 보내는 것이었다면,
클라이언트는 그 반대로 메세지를 보내고 에코 되어서 돌아온 메세지를 받는 순서로 진행하면 된다.
우선 보낼 메세지를 입력받아 byte의 배열로 인코딩 해준다.
인코딩한 배열을 서버에게 전송한다.
그 후 받는 과정은 서버에서와 똑같이 진행하면 된다.

다시 메인으로 돌아와 통신하는 부분을 방금 만든 부분으로 수정해주면 에코 서버와 통신하기 위한 클라이언트가 완성된다.

4. 프로젝트 디버깅

한 컴퓨터로 두 개의 프로젝트를 실행시켜야 하기 때문에 솔루션 설정을 바꿔줘야 한다.
솔루션을 우클릭하여 속성창을 열어준다.
[공용 속성] / [시작 프로젝트] 탭으로 이동하면 '한 개의 시작 프로젝트' 로 체크되어있을 것이다.
이를 '여러 개의 시작 프로젝트' 로 변경해주고 Client와 Server 두개 모두 '시작' 으로 설정해둔 뒤 적용시켜준다.

그다음 디버깅하여 프로젝트를 실행시켜보자
문제 없는 코드라면 'Server opened on port : 8081', 'Client joined the server' 라 출력 된 서버의 창 하나와,

'Success to join server' 라고 출력 된 클라이언트의 창이 뜰 것이다.
클라이언트 창에 메세지를 입력하여 메세지가 서버에 보내지고, 에코되어 돌아오는 것을 확인했다면 에코 서버가 올바르게 작동하고 있는 것이다.

4. 문제점

눈치가 빠른 사람이라면 프로젝트를 디버깅 해보며 이미 눈치챘을 수도 있다.

현재 서버의 구조는 다음과 같이 이루어져 있다.

1. 클라이언트가 서버에게 메세지 전송
2. 서버 메세지 처리 (출력 & 에코)
3. 서버가 클라이언트에게 메세지 전송
4. 클라이언 메세지 처리 (출력)
5. 반복

이렇게만 보면 무엇이 문제인이 알아차리기 어려울 수 있다.
그럼 좀 더 세세하게 나눠보자.

1. **서버 무한 대기 시작**
2. 클라이언트가 서버에게 메세지 전송
3. **클라이언트 무한 대기 시작**
4. 서버 메세지 처리 (출력 & 에코)
5. 서버가 클라이언트에게 메세지 전송
6. **서버 무한 대기 시작**
7. 클라이언 메세지 처리 (출력)
8. 반복

서버와 클라이언트 모두 무한 대기중인 상황이 생긴다.
아직까지도 무엇이 문제인지 모르겠다면 서버 개발자를 그만두던, 게임 개발자를 그만 두던 미래에 대한 생각을 다시 한 번 해보길 바란다.

무한 대기에 걸리는 것이 왜 문제인지 모르겠다면 다음과 같은 상황에 대입해보면 쉽게 이해할 수 있다.
경매장에 현물 100만원짜리 아이템을 올리기 위한 상황이라고 가정해보자.
경매장에 등록하기 위해 아이템을 인벤에서 없앤 후 서버에 요청을 보낸 상황이다.
하지만 서버가 다른 클라이언트와 통신 중이기에 무한 대기 상태에 빠져 우리의 요청
즉 피와 땀과 노력의 산물인 현물 100만원짜리 아이템을 받지 않는다고 해보자.
그럼 우린 언제 우리의 차례가 돌아올 지도 모른 체 계속해서 대기하는 것이다.
계속. 계ㅖㅖㅖㅖㅖ속.
그러다가 서버가 꺼지기라도 한다면?
뭐 별 다른 방법이 있나. 그냥 사라지는 것이다.

이런 최악의 상황을 제외 하더라도
한가지 요청을 처리할 때 까지 다음 요청이 지체 된다면
모든 플레이어들의 움직임 요청을 처리해야 하는 FPS 게임과 같은 경우
매 순간 모든 플레이어들이 브레이킹 댄스를 추는 광경을 볼 수 있을 것이다.

따라서 서버는 절대 멈춰선 안 된다.
저어어어어어어얼대
그럼 어떻게 해야할까?

5. 해결 방법

이 때 '멀티스레딩' 이라는 친구가 등장한다.
자세한 건 다음에 알아보도록 하고 스레드 하나가 일꾼 하나라고 알고있도록 하자.
기존의 서버는 하나의 스레드 즉 하나의 일꾼으로 작동하고 있던 것이다.
따라서 작업을 처리하는 하는 중엔 다른 작업을 처리할 수 없기에 대기 상태에 빠져 있던 것이다.

그럼 일꾼이 여러개라고 가정해보자.
하나의 일꾼은 클라이언트의 접속과 접속 해제를 담당하고,
다른 일꾼들은 각각 메세지 처리, 송신과 수신 이렇게 분업하여 담당한다면 일꾼이 무한 대기하는 상황은 벌어지지 않을 것이다.
그렇게 된다면 당연히 앞서 예시로 들었던 해괴한 상황 또한 예방할 수 있다.

멀티 스레딩은 당연히 쉽지 않다.
지금까지 개발자 친화적 코드를 짜 왔다면 이젠 CPU 친화적 코드를 짜야하고
예상하지 못한 일들이 마구 벌어질 것이다.
이런 난관을 버텨내고 멀티스레딩이라는 장벽을 넘어서 멀티스레딩 서버를 개발할 수 있는 실력이 된다면
개쩌는 서버 개발자로서 거듭날 수 있다.