서버 기술 문서 / 김정헌 / 010 9559 6056 / lapulass@naver.com / http://blog.naver.com/lapulass

목차

1. 프로젝트 소개

A. 제목: KaveOver(KPU+ cave over)*학교 이름인 'KPU'와 '뒤집히다'의 cave over의 합성

B. 장르: MMORPG + TPS *당시 화제작(The Division) 모방

C. 동기: 학교 장기 프로젝트 졸업 작품

D. 개발기간: 5개월

E. 개발 인원: 4 명

F. 맡은 역할: 서버, 팀장, 클라이언트 2D, U.I, 동기화

2. 서버 내 오브젝트

A. MAP 크기: 2km x 2km

B. 충돌 오브젝트 개수: 약 3만개(집, 자동차, 나무 등)

C. 적 NPC: 4000

D. 까마귀: 300

3. 서버 전체 흐름도

A. 충돌 체크 MAP과 적 NPC들의 정보 LOAD, A Star Path 생성

B. 네트워크 초기화 시작(IOCP 모델)

C. Thread 생성

i. Thread는 Accept, Timer, Worker Thread 5개 (*당시 서버 하드웨어가 듀얼 코어 CPU의 하이퍼 쓰레딩 2개에 예비로 1개까지 총 5개를 선택)를 생성 한다.

ii. Worker Thread는 패킷의 송수신을 담당하며, 수신된 패킷을 조립하고 처리하는 일을 한다. 패킷 처리 함수 안에는 필요한 부분에 lock을 걸거나 Lock free 컨테이너를 사용하고, 읽기만 하는 부분은 초기화하는 부분에 만들어놓아 읽기만 하

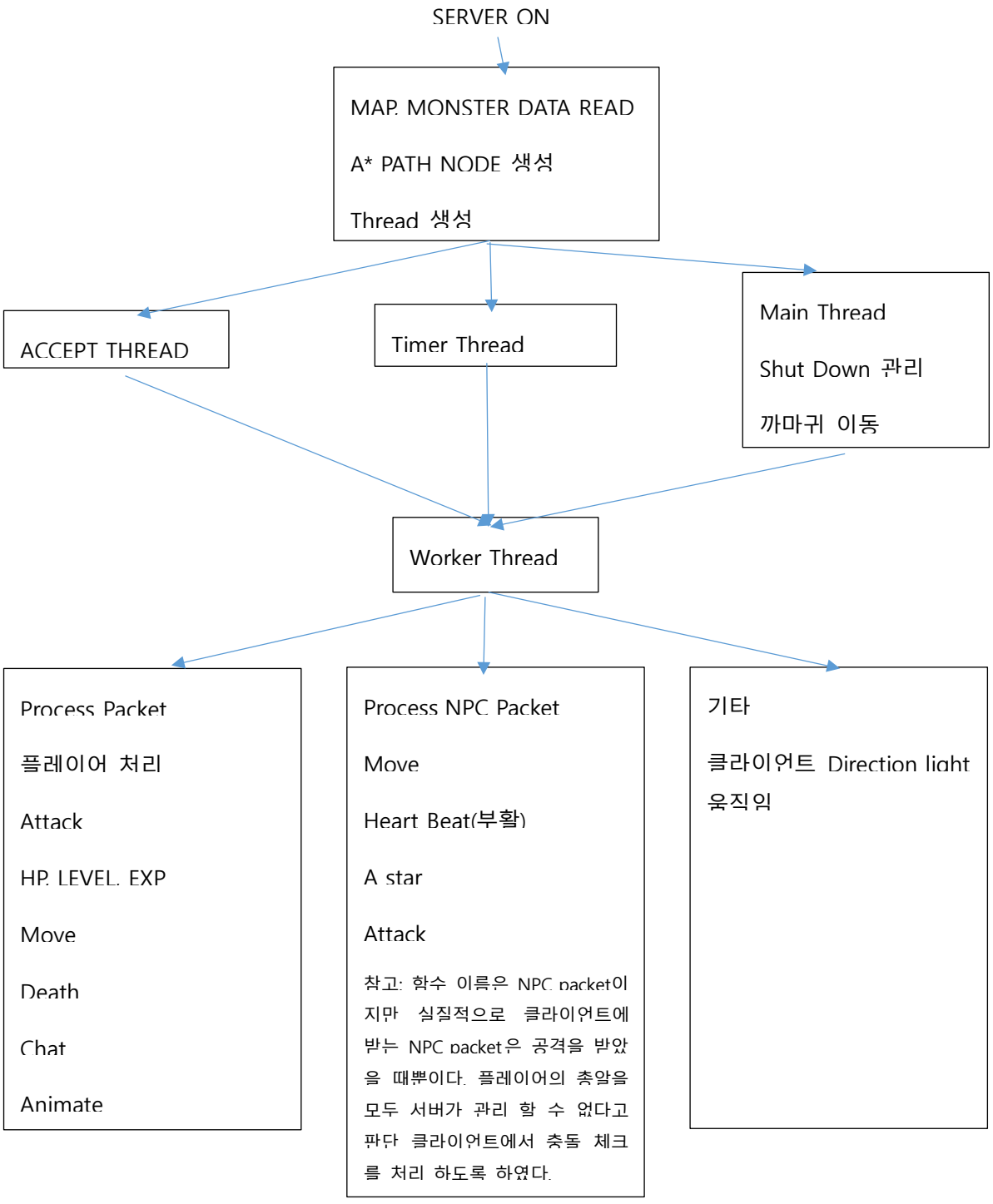
여 원자성을 보장하였다. 패킷 처리가 끝나기 전까지 다시 receive 하지 않기 때문에(N send: 1 receive) Move와 같은 경우 플레이어의 위치를 쓰는 거지만 동기화하지 않았다. 쓰는 것이 별로 없고 읽기가 많은 부분은 별로 없기 때문에 RWLock은 사용하지 않았다 자주 쓰고 자주 읽는 부분은 아래 플레이어 시야 처리에서 설명한다. 서버의 코어 부분은 혼자 하여 동기화하는 부분에 대해서 내가 알고 있기 때문에 문제되는 점은 없었지만 만약에 다수의 인원이 작업하는 것이라면 lock에 대한 고민을 보다 더 많이 해야 할 것이다. 현재는 직렬화에 대해 공부하고 있는 중이다.

iii. Accept Thread는 Listen 소켓을 생성하며 클라이언트들의 접속을 받고 초기화 작업(위치나 보이는 플레이어들이나 NPC들의 위치 정보들)들을 담당한다. Accept를 따로 뺏기 때문에 클라이언트들의 접속이 없을 시에는 Thread가 놀게 된다. 당시에는 AcceptEX 함수의 사용법을 자세히 알지 못해 활용하지 못하였다. 현재 함수포인터 등을 활용해 AcceptEX와 더불어 확장 함수들에 대한 서버를 제작해보고 있다.

iv. Timer Thread는 NPC들이 시간에 의해 행동하는 것을 하나의 컨테이너의 담고 시간에 맞춰 꺼내 쓰는 Thread이다. NPC들은 플레이어에 시야에 들어 올 때만 행동하므로, 한 명의 플레이어도 NPC가 보이지 않는다면 쉬는 Thread이나 게임의 구조상 NPC가 보이지 않는 곳은 없다. 따라서 쉬지 않는다. 호모지니어스의 방식에서 조금 어긋나지만 그렇다고 해서 이 Thread만 바쁜 것은 아니기에 Timer Thread를 따로 두는 것을 선택 하였다.

v. Main Thread는 초기에는 서버의 종료와 함께 서버에 생성 된 것들을 정리하는 것으로 구상하였으나, 개발 말기에 졸업 작품 발표를 앞두고 가장 놀고 있는 Main에 까마귀 군집 이동을 넣었다. 따라서 현재 서버는 강제 종료 외에는 서버를 안전하게 끄는 방법은 없다. 서버의 안정성이 많이 떨어지지만 당시 게임 제작에 있어서는 게임 플레이에 대한 데이터보다 게임의 규모(맵 크기, 적 NPC 수 등)이 더 중요 했기에 이런 선택을 하였다.

D. Process Packet과 NPC는 switch case로 작성되어 굉장히 길고 복잡하다. 그리고 전역함수로 나와 있기 때문에 계속해서 Server Class 안에 있는 객체들을 불러들인다. 포인터를 이용해 주소 값만을 불러들이는 거지만 클래스 안에서 했다면 그마저도 줄어들어 성능에 조금은 도움이 됐을 것이다. 포트폴리오에 아주 간단하게 함수포인터를 이용해 Process Packet의 길이를 줄여 함수만 추가하여 따로 작업 할 수 있는 방향으로 만들어보았다. 이 방법에 경우 함수를 따로 건들고 수정 사항이나 추가 사항이 눈에 보기 좋게 되어 있어 협업에 좋을 것이라 예상한다.


SERVER ON
MAP. MONSTER DATA READ
A* PATH NODE 생성
Thread 생성
ACCEPT THREAD
Timer Thread
Main Thread
Shut Down 관리
까마귀 이동
Worker Thread
Process Packet
플레이어 처리
Attack
HP. LEVEL. EXP
Move
Death
Chat
Animate
Process NPC Packet
Move
Heart Beat(부활)
A star
Attack
참고: 함수 이름은 NPC packet이지만 실질적으로 클라이언트에 받는 NPC packet은 공격을 받았을 때뿐이다. 플레이어의 촘알을 모두 서버가 관리 할 수 없다고 판단 클라이언트에서 충돌 체크를 처리 하도록 하였다.
기타
클라이언트 Direction light
움직임

4. NPC 흐름도

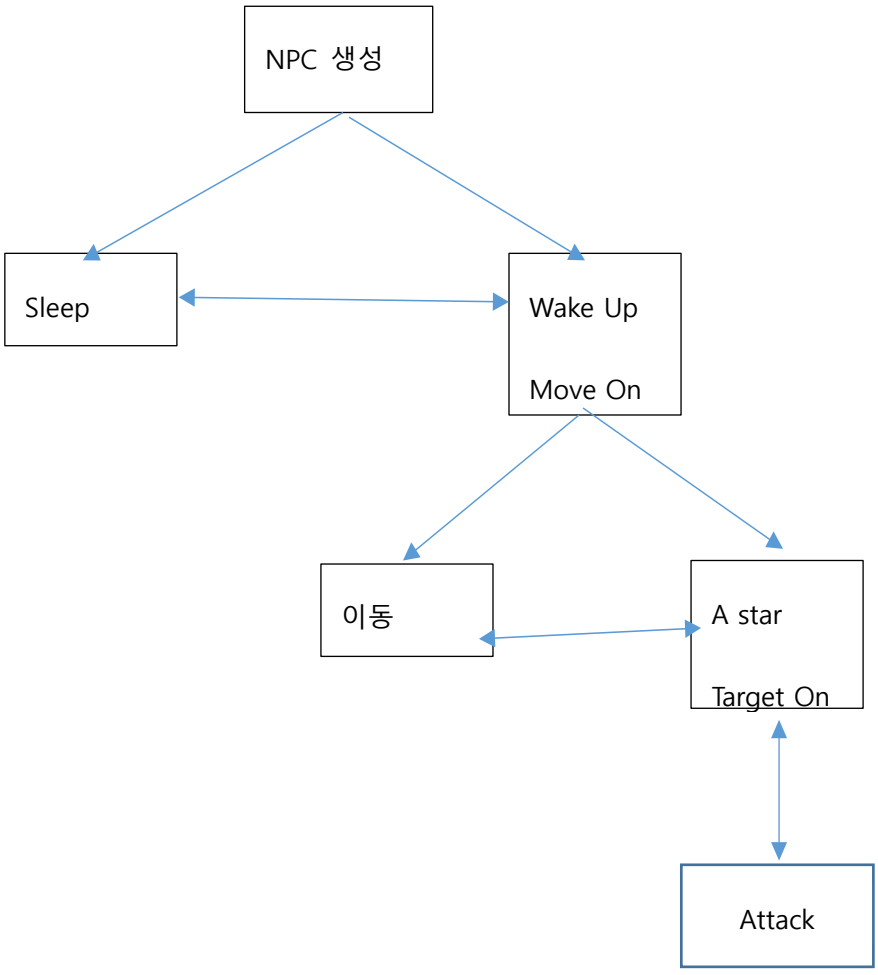
A. NPC들은 초기 생성 시 모두 Sleep 상태가 된다. 플레이어의 지정한 시야 만큼 들어오면 Wake Up하여 Timer Thread를 통해 이동 한다.

B. NPC들은 3가지 상태를 가지는데 이동, A star, Attack이 있다. 이동은 8방향으로 랜덤하게 움직이며 적정거리에 들어온 플레이어를 A Start를 통해 따라가게 된다. 일정거리에 안에 들어오면 Attack을 행하게 된다.

i. 이동은 적 NPC가 깨어나게 되면, 다른 플레이어에게도 Wake Up 하지 않도록 깨어났다고 표시를 하여 동기화를 시도해 보았다.

ii. A star도 마찬가지로 한 플레이어를 따라가게 되면 다른 플레이어를 따라가지 않도록 동기화 해두었다. 플레이어가 지정한 거리보다 멀어지거나 다시 Sleep 상태가 되면 다른 플레이어에게 깨어나거나 따라가게 된다.

iii. Attack은 A star에 묶여 있고 단일 공격이기 때문에 A star와 마찬가지로 한 명의 플레이어만 공격 한다.



5. 기술 소개

A. 플레이어 시야 관리

i. 플레이어들을 동기화하기 위해 시야를 이용한 컨테이너(이하 시야 리스트)를 사용 했다. 플레이어들은 각각 자신이 보이는 플레이어를 자신의 컨테이너의 담고 보이는 플레이어에게만 자신의 정보들을 보낸다. 이렇게 한 이유는 처음 보이는 플레이어인지 아닌지를 판단하기가 어렵다. 처음 보이는 플레이어면 클라이언트에게 지금 보이니 그림을 그려라! (Draw call) 라고 보내야 하는데 거리 혹은 나눠진 지역 정보로는 그것을 판단하기 어렵다고 생각 했기 때문이다. 시야 리스트를 사용 할 때 동기화가 필요한데 처음에는 mutex를 이용하였다. 커널 객체인 뮤텍스를 이용한 이유는 사용하기 편해서였고, 보다 빠른 유저 모드인 Critical section이나 interlocked 계열에 busy waiting 관리와 따로 함수를 만들어 줘야 하는 번거로움 때문이었다. 서버 안 코드를 자세히 보면 뮤텍스를 사용하는 곳이 별로 없는데 이러한 이유는 2가지가 있다.

첫 번째는 다중 lock이다. 플레이어 하나의 두 마리의 적 NPC가 같은 시야 리스트를 사용 하기 때문에 우연히 동시에 두 마리가 깨어 났다면 이중 lock이 걸린다. 이때 같은 뮤텍스 객체의 lock을 시도하기 때문에 오류를 뱉어낸다. 그때 해결 했던 방법은 recursive lock을 사용하여 unlock만 숫자가 맞으면 다중 lock을 사용 할 수 있다.

두 번째 위의 문제를 해결 하여도 다수의 플레이어와 다수의 적 NPC가 만나게 되면 너무 많은 lock들이 난무하기 때문에 서버가 느려지게 되고 오랫동안 멈춘 것처럼 되어버린다. (처리를 하고 있으므로 정확히는 멈춘 것은 아니기 때문에 '것처럼' 이라고 표현했다.)

따라서, Lock free 알고리즘을 사용 하였고 이 문제를 어느 정도 해결 하였다. 어느 정도라고 표현한 것은 메모리 누수 관련하여는 해결하지 못했기 때문이다. List는 메모리 재활용을 해결 하였지만 Skip List에 경우는 계속해서 해보고 있지만 아직 성과는 없다.

B. 플레이어의 충돌체크

i. 행렬의 수학 공식을 이용해 공간을 분할 하였다. 수학 공식 중에 행렬의 위치를 빨리 찾는 공식이 있기 때문에 플레이어의 위치를 통해 그 행렬을 찾게 되고 그 번호 안에 있는 오브젝트들만 충돌 체크를 하였다. 충돌 체크는 AABB로 하였는데, 정적인 객체는 움직이지 않으므로(회전하거나 하는 오브젝트가 없다.) 빠른 충돌 체크 방법을 선택 하였다. 참고로 클라이언트에도 이 공간분할을 사용 했

다.

C. A star

i. 서버에서 A star를 통해 적 NPC들이 길 찾기를 시도한다. A star를 사용한 이유는 학교 졸업 작품에서 높은 난이도의 알고리즘으로 평가 받고 있기 때문이었다. 우리 게임은 격자 형식의 MAP이기 때문에 A star보다는 바로 바로 길을 찾거나 배열로 바닥에 true false만을 표시해 길을 찾는 것이 훨씬 더 나은 방법이지만 졸업 작품의 난이도를 위해 A star를 사용하였다.

D. Flock

i. 서버에서 Flock 알고리즘을 통해 클라이언트 내의 하늘에 있는 까마귀들이 군집 이동을 하게 된다. 사용한 이유는 위에 A star와 같이 졸업 작품의 난이도를 상승 시키고, 무언가를 많이 했다라는 걸 표출 하기 위해 사용 하였고 게임 초기 기획에는 없었다. 당시 서버에서 콘텐츠를 추가하고 싶어도 클라이언트의 개발이 밀려 있어 Direct 3D를 배운 내가 빠르게 추가 할 수 있는 기하 쉐이더를 이용해 까마귀들을 클라이언트 내에 띄우고 Flock 알고리즘을 사용했다.

6. 로그인 서버 설계

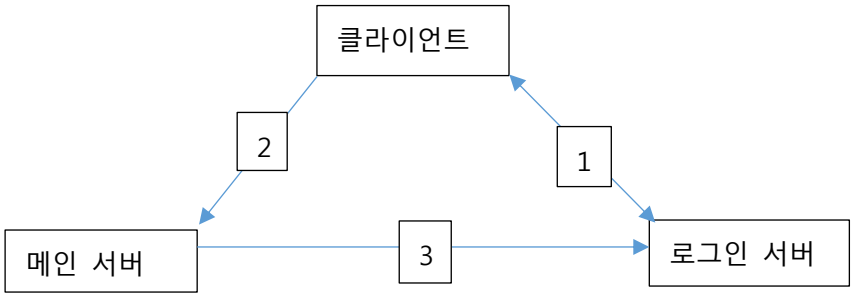
A. 졸업 작품 서버 코드를 보면 //ksh라는 주석이 있다. 이 부분은 내가 아닌 다른 팀원이 작성한 부분이고, 로그인 서버를 연결 하려고 시도한 부분이다. 졸업 작품 발표 전까지 완성하지 못해서 주석처리를 해놓았다. 기술 문서에 작성하는 이유는 로그인 서버(DB 서버)와 메인 서버의 연결하는 것에 대한 설계를 했고 초반 부분을 작성하여 되는 것을 확인하고 넘겼기 때문이다. 그 이후에 로그인 서버에 대한 것은 팀 원에게 맡겼다.

B. RPG 서버에서는 로그인 로그아웃을 Accept와 플레이어가 종료 했을 시에 처리 하도록 하였다. RPG 서버는 개인 프로젝트에 개발 기간이 짧아 서버를 따로 둘 수 없어 그렇게 시도 하였다. 그렇게 했을 때 문제점이 더미 클라이언트들이 접속하거나 접속 종료를 했을 때 DB의 접속하거나 쿼리를 보내는 것(저장 프로시저를 통하여 했음에도 불구하고)에 많은 양이 몰려 생각했던 것 보다 접속도 느리고 접속 종료 했을 때에도 문제가 많았다. (더미 클라이언트는 보통 동시에 접속해서 동시에 접속 종료를 하기 때문) Accept는 따로 Thread 하나로 돌리기 때문에 싱글 Thread와 다름 없어 접속하는 것이 느리고(초기화 작업이 많다면 더욱 더) 접속 종료 시에도 Worker Thread는 5개 밖에 존재 하지 않고 더미 클라이언트가 접속 종료 할 때는 보통 몇 백 개가 한 번

에 나가므로 문제가 생긴다. 이러한 문제들로 졸업 작품 서버에서는 로그인 서버를 두기로 하였다.

C. 졸업 작품 서버에서도 마찬가지로 로그인(회원 가입)과 로그아웃만 존재 하도록 기획 하였다. 플레이 중간 중간에 사용하지 않은 이유는 로그인 서버의 설계와 초반 코어 부분만 도움을 주고 나머지는 상대방에게 맡겼기 때문에 메인 서버를 건들다가 서버 자체가 문제가 생길 것을 염려 했다. (파티 시스템도 맡겼었다. 졸업 작품 서버 코드 내에 Process Packet 부분에 //ksh가 존재하지만 문제가 많아 전부 주석 처리를 했다.)

*만약 플레이 중간에 DB에 접근 할 일이 생기면 DB 저장 순서에 대한 보장을 추가로 생각해 보아야만 한다.



i. 1번 클라이언트가 로그인 서버로 접속 한다. 로그인 서버는 Accept때 DB가 이 플레이어가 있는지 없는지를 판단 한다. 로그인 서버에서 DB에 접속 해 확인 하여 메인 서버로 접속을 시켜 준다. 이때도 마찬가지로 Accept Thread는 하나 이기 때문에 다수의 접속이 있을 때 문제가 생긴다. B에서 언급한 로그인 문제는 해결하지 모하였다. 이 문제는 위에서도 언급 했듯 당시에는 AcceptEX를 제대로 사용하지 못하였다.

ii. 2번 클라이언트와 메인 서버가 상호 작용 한다.

iii. 3번은 로그아웃 시에 발생 한다. 클라이언트가 접속을 종료하면 메인 서버는 그 정보를 로그인 서버에 보내준다. 메인 서버는 초기 네트워크 작업을 할 때 로그인 서버와 연결 되어 있는 상태로 있는다. 만약에 플레이 중간에도 DB에 저장 할 일이 생기면 로그인 서버에 정보를 보내 주면 된다. 로그인 서버는 IOCP로 제작 되어 있는데 당시에는 메인 서버는 내가 제작하고 있기 때문에 IOCP를 다룰 일이 별로 없는 팀 원에게 연습 겸 IOCP로 제작하라고 했었다. 하지만 로그인 서버가 정보를 받는 건 초기에 Accept를 제외하고는 메인 서버에게 밖에 받지를 않는다. 따라서 메인 서버에게 받는 패킷만 처리하는 Thread 하나만 필요하다. (receive는 패킷 처리하는 동안 1번 밖에 일어나지 않기 때문. 당시에는 그랬지만 AcceptEX를 쓰면 Worker Thread가 필요하다. 또 서버 규모가 커지면 메

인 서버 하나만으로 안되기 때문에 다양한 서버(게임 서버와 같은)에서 패킷을 받을 수도 있다.)

이렇게 하면 메인 서버는 로그아웃 했을 때 패킷 정보만 송신하면 되므로 성능이 보다 향상이 된다.

7. RPG 서버 추가 사항

RPG 서버는 졸업작품 서버를 만들기 전 게임 서버와 IOCP에 대해 기초 지식을 쌓았던 서버이다. 물론, 그 전에 비동기 모델 없이 만든 크레이지 아케이드 모방 게임이 있지만 최대 유저 수가 4명밖에 되지 않는 규모가 보다 작은 서버이다. 졸업작품 서버는 RPG 서버에서 조금 더 발전 된 상태라 할 수 있겠다. 따라서, 졸업 작품에서 없는 부분만 설명하도록 하겠다.

A. 데이터베이스

RPG 서버에는 내가 직접 연결한 하였다. 로그인, 로그아웃에서 DB에 접근하도록 하였다. SQL과 DB는 MySQL로 공부하였고, 서버에는 MSSQL을 연결하였다. MSSQL을 연결한 이유는 MSDN과 같이 참조 자료가 보다 많기 때문이다. 데이터베이스의 스키마는 ID와 위치 정보, 레벨, 경험치를 저장하며 클라이언트가 Accept시 DB에서 확인 하며 클라이언트의 종료의 루틴에서 갱신해준다.

B. LUA 스크립트

몬스터의 움직임에 스크립트를 적용하였다. 복잡한 계산은 LUA를 사용 할 시 성능이 보다 떨어지므로, 복잡한 계산은 C 언어 함수를 불러와 계산하는 방법을 선택하였다. 개인 작업이라 스크립트에 필요성이 많이 떨어져 많은 부분을 넣지는 못하였다.

8. IOCP 채팅 서버

A. 채팅 서버를 만든 목적은 기존 서버에 부족한 부분을 보안하기 위해서이다. 최종 목표는 호모 지니어스에서 벗어난 Accept 쓰레드를 작업자 쓰레드 안으로 집어 넣고, Process Packet을 협업에 용이한 직렬화 방식에 대한 고안이다. 따라서, 채팅은 기존 서버에 관련한 책에 많이 나오는 방법이라 설계하기 쉽기 때문에 선택한 것이고 목적은 위에 언급한 기존에 내가 갖고 있는 서버보다 나은 것을 연구하기 위함이 목적이다.

B. 기존 서버에 난잡한 파일 분할을 조금 더 읽기 쉽게 바꾸어보았다.

C. 함수 포인터를 이용해 Process Packet을 추가와 찾기가 눈에 더 빨리 들어오도록 바꾸어보았다. 협업 할 시 유용 할 것 같다.

D. 기존 서버에 종료 루틴이 없는 부분을 추가 하였다.

E. 기존 서버에 전역 함수로 존재하던 쓰레드와 패킷 처리 부분을 클래스 안으로 넣어 멤버 변수에 접근 하기 용이하도록 바꾸었다. 기존에는 주소 값을 꺼내 써야 하는 번거로움을 없애 줄 수 있었다.

F. 더미 클라이언트를 통해 현재까지 작성 된 부분이 제대로 돌아가는 것을 확인해 보았다.