네트워크 엔진

Network Engine

with IOCP

2021년 6월

노 영 래

목 차

제 1 장 서론

네트워크 파트의 전반적인 부분을 도맡아서 하는 네트워크 엔진을 구현하고자 한다. 사용자는 컨텐츠를 구현한 뒤 해당 네트워크 엔진의 객체를 선언하고 Start 함수만 호출해주면 된다. 네트워크 엔진은 IOCP(Input/Output Completion Port)를 사용하기 때문에 윈도우 서버 위에서 돌아가는 것을 전제로 하며 공유자원이 비교적 적어 IO WorkerThread에서 독립적으로 일을 처리할 수 있는 컨텐츠를 위한 버전(로그인, 채팅)과 게임과 같이 공유자원이 많아 IO WorkerThread에서 독립적으로 처리할 수 없는 버전(동기화 객체를 사용하면 되지만 여기서는 최대한 지양하고 있다.) 두 가지 버전을 구현하고 성능분석을 진행했다.

제 2 장 구성

2.1 Memory Pool

2.1.1 Simple Memory Pool

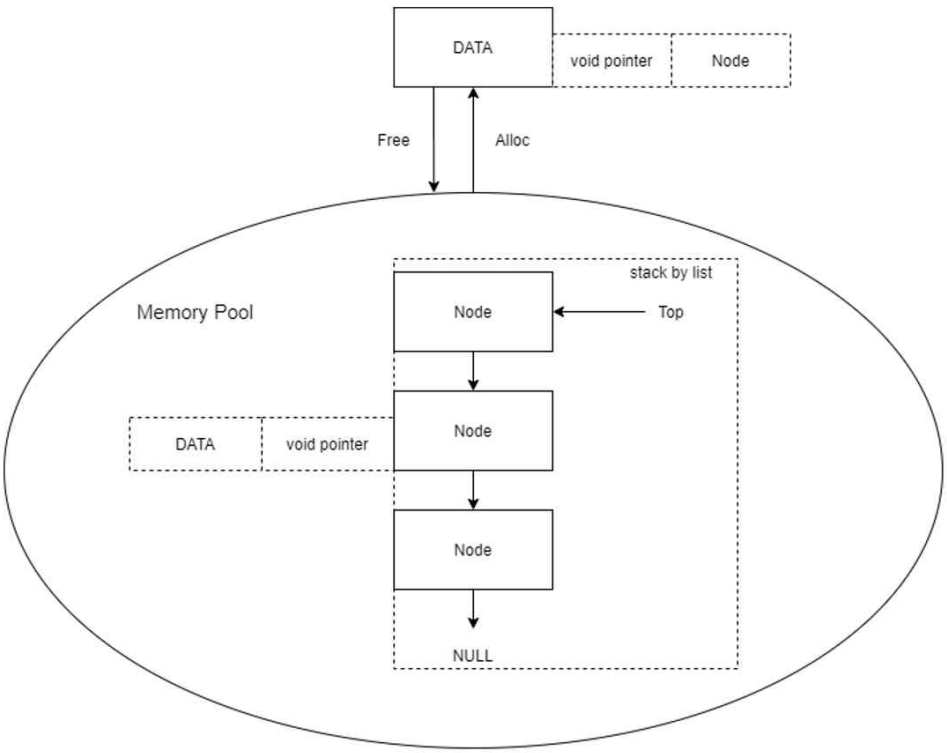

그림 1 Simple Memory Pool

new & delete 함수는 무겁기 때문에 고성능의 네트워크 엔진을 만들기 위해서 Memory Pool의 사용은 선택이 아닌 필수다. Memory Pool이긴 하지만 처음부터 모두 할당해 놓고 사용하는 것이 아닌 Free List 방식을 사용했다. 사용자가 Alloc을 요청했을 때 stack에 노드가 있다면 Top의 노드 앞에 달려있는 DATA의 포인터를 반환해주면 되고 없다면 DATA+void pointer+Node 크기만큼 할당하여 DATA의 포인터를 반환해주면 된다. 사용자가 Free를 요청하면 DATA 뒤쪽에 있

는 Node를 참조하여 스택에 넣어주면 된다. void pointer 자리에는 할당 시 Memory Pool Pointer(this)를 넣어주고 해제시 this와 같은지 확인하여 여기서 할당한 것이 아닌 포인터의 Free를 요청받지는 않았는지, 메모리를 잘못 참조하여 사용하지 않았는지를 확인할 수 있다.(heap 영역에 메모리를 over wirte하면 생성자나 소멸자 호출 시 에러를 뱉는 것처럼 에러를 뱉게 했다.)

여기까지만 구현하면 template의 type이 클래스일 경우 부족하다. 우리가 new를 호출할 때 단순 메모리만 할당받아오는 것이 아니다. 생성자도 내부에서 호출되고 있다. 그래서 placement new를 사용하여 Alloc 시 생성자를 호출, Free 시 소멸자를 호출해주는 버전을 플래그에 따라 분기를 타게 했고 해당 플래그는 Memory Pool 생성자 에서 인자로 받도록 했다.

2.1.2 LockFree Memory Pool

위 구조는 멀티스레드 환경에서 안전하지 않다. 일반적으로는 동기화 객체를 사용하여 Alloc Free를 할 때마다 Lock을 걸어주면 되지만 여기서는 Memory Pool에서 사용하고 있는 스택을 Lock Free 구조로 만들어서 동기화 객체를 사용하지 않는 구조로 만들었다.

Alloc Free를 할 때 stack에 변화를 주는 것은 top을 옮기는 행위다. top을 임시로 저장해 놓고 노드 작업을 한 뒤에 임시 저장된 top이 현재 top과 같다면 top을 옮기고 다르다면 과정을 다시 반복한다. (일단 해보고 그 동안 변화가 없었다면 commit을 하는 방식) top을 비교하고 바꾸는 행위가 원자적으로 일어나야 하기 때문에 InterlockedCompareExchange를 사용한다. 하지만 top의 포인터만으로는 변화를 감지할 수 없는 경우도 있기 때문에 Alloc Free를 할때마다 넘버링을 하여 InterlockedCompareExchange128을 통해 top의 포인터뿐만 아니라 기억된 숫자도 같이 확인하여 변화를 탐지한다.

2.1.3 TLS Memory Pool

위 구조는 멀티스레드 환경에서는 안전하지만 경합(top이 계속 변화하기 때문에 commit을 하려고 계속 반복문을 도는 경우)이 너무 많이 일어나기 때문에 단순 저 구조만으로는 성능의 관점에서 멀티 스레드 환경에서 사용할 수 없다.(그것은 동기화 객체를 써도 마찬가지다) 그래서 TLS를 사용하여 성능을 개선했다.

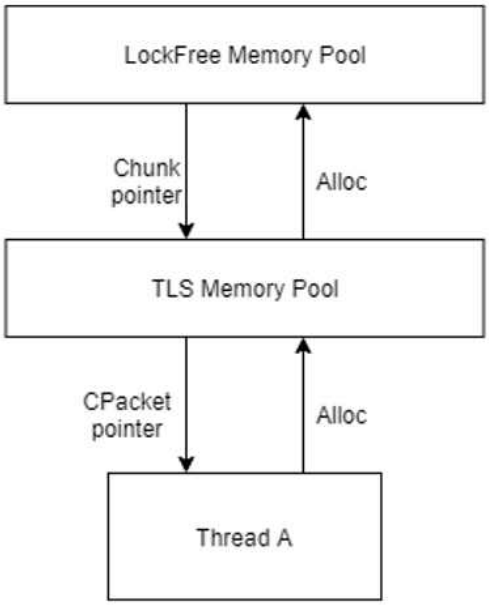

그림 2 TLS Memory Pool

뒤에서 사용할 패킷 클래스를 예로 설명하겠다.

1. 우선 TLSMemoryPool 객체를 선언해야 한다. TLSMemoryPool 생성자에서는 tls 값을 할당받고 Chunk 내부의 CPacket 배열 크기를 인자로 받을 수 있다. Chunk 는 CPacket 배열의 포인터와 AllocCount와 FreeCount를 맴버로 가지고 있다.

```
TLSMemoryPool<CPacket> packetPool(1000);
```

2. TLSMemoryPool<CPacket>은 LockFreeMemoryPool<Chunk>를 포함하고 있다. packetPool을 참조하여 Alloc을 할때 최초(TlsGetValue return NULL) 그리고 Chunk를 모두 사용했을때는(AllocCount==ChunkSize) LockFreeMemoryPool로부터 Chunk를 할당받아 포인터를 Tls에 셋팅하고 이에 해당하지 않는 경우(대부분의 경우)에는 그냥 Tls로부터 Chunk pointer를 가져와 참조하여 CPacket 배열에 접근해 AllocCount를 index로 해당 배열 요소의 포인터를 가져온다. 그리고

AllocCount를 증가시킨다.

3. LockFreeMemoryPool 내부에서 Chunk 생성자를 호출시키는 경우(최초 할당의 경우다. 할당할때마다 생성자를 호출할 필욘는 없다.) 단순히 CPacket 크기만큼 배열을 할당하는 것이 아니라 CPacket과 Chunk Pointer의 크기를 합한 크기만큼 배열을 할당해야 한다. 그리고 모두 해당 Chunk 객체의 pointer를 집어 넣어 놓는다. 이렇게 하는 이유는 사용자가 패킷을 다 사용 후에 해제를 요청했을 때는 할당 때와 달리 특정 스레드에 종속되지 않는다. (어디서 해제될지 모른다.) 그렇기에 패킷으로부터 자신의 Chunk를 찾을 수 있게 하기 위해 ChunkPointer도 같이 저장하는 것이다. AllocCount에는 0을(0번 index부터 값을 올려가며 사용자 요청시 CPacket의 포인터 반환) FreeCount는 MaxCount가 지정된다.

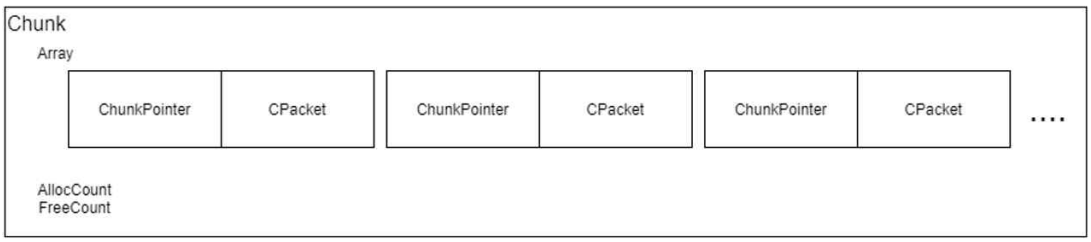

그림 3 Chunk

4. 사용자가 해제를 요청할 경우 TLSMemoryPool의 Free에서는 인자로 받은 패킷 주소로부터 8byte이전 주소의 값을 참조에 Chunk pointer에 접근할 수 있다. 해당 Chunk 객체의 FreeCount를 1 감소시키고 0이 되면 해당 Chunk 객체는 모두 사용되고 해제되었으므로 LockFreeMemoryPool에 해제를 요청한다.

정리하면 멀티스레드 환경에서 서로 다른 스레드가 LockFreeMemoryPool를 사용해 메모리 할당 해제를 할 시에 부하가 걸리면 성능에 크게 영향을 줄 수 있다. 그러므로 덩어리로 할당받아 가져와 TLS에 저장해 놓는 방식으로 경합 없이 할당 해제를 하고 해당 덩어리를 다 할당하고 다 해제 시에만

LockFreeMemoryPool에 할당 해제를 요청을 하는 방식을 사용했다. 이렇게 하면 실제 LockFreeMemoryPool에 요청하는 빈도를 대폭 낮추어 스레드간 경합을 최소화 할 수 있다.

2.2 LockFreeQueue(SendQ)

멀티스레드에 안전한 큐는 다양한 곳에서 필요하다. 네트워크 엔진 내부에서는 Send를 위해 데이터들을 모아 놓을 큐가 필요하고 서로 다른 스레드에서 이 큐에 enqueue를 할 수 있기 때문에 LockFreeQueue로 구성했다.

구현 방식은 위에서 MemoryPool를 만들기 위해 구현한 LockFreeStack 구조와 비슷하다. 먼저 작업을 한 후 변화가 감지되지 않았다면 commit을 하고(head나 tail을 옮기는 작업) 변화가 감지되었다면 작업하는 과정부터 다시 반복한다. 변화가 감지되지 않아 commit이 될 때까지 반복한다. 또 LockFreeQueue 내부에서는 Node를 new delete하는 것이 아닌 위에서 만든 MemoryPool을 사용한다.

여러 스레드에서 해당 세션에 보낼 것이 있다면 해당 세션의 SendQ에 Enqueue를 하고 실질적으로 Send를 할때는 SendQ에 있는 패킷 포인터들을 WSABuf에 차례로 셋팅 한다. 보낼 것을 BinaryBuffer에 모아서 한 번에 보내는 구조가 아닌 이렇게 산개한 패킷들의 버퍼를 WSABuf에 여러개 셋팅 하여 Send 하는 경우에는 비동기 IO 구조로 가면 안 된다.(비동기 IO 구조로 가는 방법은 간단하다. socket option에서 송신 버퍼의 크기를 0으로 셋팅하면 된다.) 비동기 IO구조로 가게 되면 l4 송신버퍼에 패킷 데이터들을 복사 하는 것이 아닌 복사 없이 커널 영역에서 내가 마련한 버퍼(패킷의 직렬화 버퍼)를 직접 사용한다. 복사가 줄어들고 병렬적으로 IO 작업이 진행되기 때문에 성능상 좋지만 커널에서 내가 셋팅한 버퍼를 직접 사용하는 구조기 때문에 내 패킷이 포함된 페이지가 (4KB) Lock이 걸린다.(여기서 Lock은 디스크로 디커밋 되지 않는 상태를 말한다.) 메모리에 너무 많은 페이지가 상주하는 것은 둘째치고 페이지 Lock을 걸 때 하나의 페이지를 대상으로 Lock을 거는 거야 거의 영향이 없는데 각기 다른 페이지에 상주한 몇백개의 패킷의 페이지를 Lock을 걸게되면 WSASend 함수 자

체의 성능에 크게 영향이 간다. 그래서 위 구조에서는 비동기 IO를 사용하지 않는다.

2.3 RecvRingBuffer

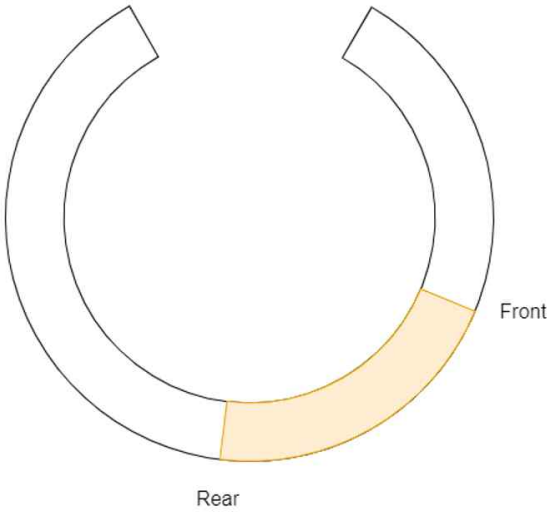

그림 4 RecvRingBuffer

WSARecv를 통해 비어 있는 RecvRingBuffer 크기를 최대로 받을 수 있는 만큼 받는다.(그림과 같이 비어있는 부분이 두동강이 나 있을 경우 두 개의 버퍼 포인터를 셋팅하여 WSARecv를 호출) 헤더는 크기가 고정이기 때문에 헤더 크기만큼 뽑을 수 있는지 확인. 뽑을 수 있다면 헤더에 명시된 payload size만큼 뽑을 수 있는지 확인. 이것도 만족한다면 데이터를 뽑아 패킷의 직렬화 버퍼에 복사한다. RecvRingBuffer를 거치지 않고 그냥 Packet의 직렬화 버퍼를 셋팅하여 WSARecv를 호출하는 방식도 있다. 이 경우 직렬화 버퍼 안에 완성된 메시지가 여러개

들어있는 경우도 생기기 때문에 이것들을 모두 처리하기 전까지는 Packet을 해제하면 안 된다.(복사량이 줄어 성능은 좋아질 수 있지만, 구조가 복잡하다)

2.4 Packet

패킷을 관리하기 위해 CPacket 클래스를 만들었다. 클래스 내부에는 다음과 같은 기능을 가진 함수들을 구현했다.

1. 인코딩 디코딩
2. 패킷 마샬링 과정을 해당 패킷 구조체로 형변환하여 맴버로 접근하는 방식이 아닌 직렬화 버퍼 방식을 사용.(가변적 프로토콜을 설계할 때 좋다.) >> << 연산자를 다양한 자료형을 대상으로 오버로딩 하여 데이터 마샬링에 편의를 주었다.

패킷 또한 위에서 구현한 MemoryPool을 사용한다. 패킷을 지역에서 선언해서 한번쓰고 버리고 다른 곳으로 복사하는 구조가 아니라 포인터를 들고 다니며 사용하다가 필요 없을 때 해제를 할 것이다. 받을 때는 위 RecvRingBuffer를 거쳐 하나의 패킷 객체에 하나의 완성된 메시지가 들어있는 구조라면 상관없다. 그냥 마샬링을 완료하고 컨텐츠가 끝나는 곳에서 해제하면 된다. 하지만 보낼 때는 두가지 경우에서 문제가 생길 수 있다. 첫째는 보낼 패킷을 SendQ에 enqueue를 하고 난 후 부터는 해제될 수 있는 곳이 여러 곳이다. WorkerThread에서 완료통지가 떨어지는 부분일 수도 있고 보내지지 않고 세션이 끊겨 세션을 초기화 하는 과정에서 SendQ에 남아 있는 패킷을 해제시켜주는 곳일 수도 있다. 이 경우

어디서 해제될지 모르는데 한 곳에서만 해제하거나 두 곳 모두에서 해제를 한다면 문제가 될 것이다. 두 번째는 하나의 패킷을 여러 세션에 broad casting을 하는 경우다. 무의미하게 패킷을 반복해서 만드는 것이 아니라 한번 만들고 여러 곳에서 사용한 후 필요없다면 해제를 할 것이다. 이러한 구조 때문에 참조카운트를 사용함으로서 해결했다. 참조 카운트를 통해 다른 스레드로 넘어갈 여지가 있는 부분에서는 참조카운트를 올리고(SendQ enqueue) 해제 가능성이 있는 부분에서는 참조카운트를 차감시켜 0이 될 때만 해제를 해 준다.(최초 시작은 0이 아닌 1에서 시작한다.) 좀 더 편의성을 중시 한다면 스마트 포인터처럼 알아서 해제되게끔 구현할 수도 있다. 클래스로 패킷포인터와 참조카운트 포인터를 맵핑하고 사용자는 이 스마트 패킷을 사용하는 것이다. 이 경우는 포인터에 직접 접근하면 안 된다. 패킷을 다른 함수에 인자로 넘길때 스마트 패킷의 복사 생성자가 호출되도록 유도하는 것이다. 복사생성자에서는 참조카운트를 올리고 소멸자에서는 참조 카운트를 차감한 뒤에 0이 되면 패킷을 소멸시키는 방식이다.(dequeue쪽에서는 값을 셋팅하고 소멸자를 명시적으로 호출시켜야 한다.) 하지만 함수 호출량이 너무 많아져 성능에 크게 영향을 주기 때문에 엔진에서 쓰는 것은 적절하지 않은 것 같다. 해당 엔진에서는 참조카운트를 수동으로 필요시에만 증가, 차감할 것이다.

제 3 장 Network Engine for Login&Chatting Server

3.1 overview

네트워크 엔진은 제공하려는 서비스에 따라 다양한 구조로 설계될 수 있다. 가장 일반적인 구조는 IO 작업을 하는 IOCP WorkerThread 에서 모든 일을(컨텐츠 까지) 끝마치는 것이다. 만약 서비스를 위한 컨텐츠들이 공유자원을 사용하지 않고 세션간 독립된 작업을 하여 공유자원에 대한 동기화가 필요하지 않다면 IOCP WorkerThread 에서 컨텐츠 파트까지 수행하는 이 구조가 가장 성능적으로도 편의적으로도 알맞은 구조다. 3장에서는 이러한 서비스를 위해 설계된 네트워크 엔진을 소개한다.

3.2 structure

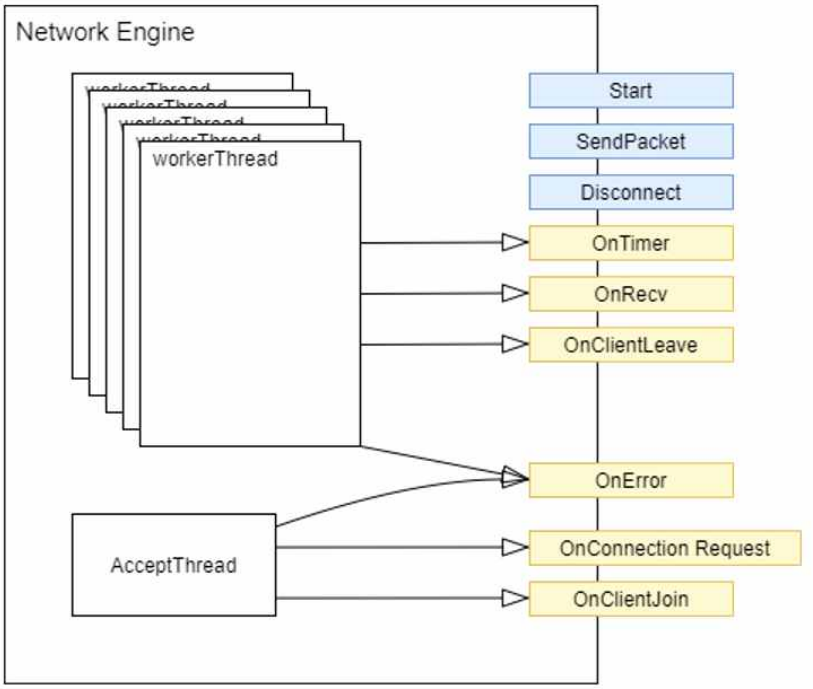

그림 5 Network Engine for Login&Chatting

위 노란색박스 안의 함수는 순수 가상함수고 파란색박스 안의 함수는 public 함수들이다. 스레드는 workerThread와 AcceptThread가 있다. (대표적인 것만 간략히 명시했다.) 사용자가 할 것은 CNetworkEngine을 상속받아 순수 가상함수들을 구현하고 해당 객체의 Start 함수를 호출해 주는 것이 전부다.(Start 함수는 인자

로 ip, port, WorkerThread의 개수, 동시에 돌아갈 WorkerThread의 개수, 세션의 최대 수 등을 넣어 제어할 수 있다.) 그럼 네트워크 엔진은 돌면서 사용자가 구현해 놓은 함수들을 호출해 준다. 에러가 발생할 때는 OnError, Accept를 한 직후에는 OnConnectionRequest(필터링 역할), 세션을 모두 셋팅해 준 후에는 OnClientJoin, RecvRingBuffer로부터 완성된 메시지를 뽑아 패킷을 생성하면 OnRecv, 연결이 끊겼다면 OnClientLeave, 일정시간 이상 메시지가 오지 않는 세션이 있다면 OnTimer 함수를 호출해 준다. 호출되었을 때 무엇을 할지는 사용자가 결정하는 것이다.

3.2.1 WSASend WSARecv

WSASend 와 WSARecv를 중첩해서 걸지 않는다. WorkerThread 내부만 복잡해 질 뿐 이점이 없다. WSASend와 WSARecv를 하고 IOCP WorkerThread의 GQCS (GetQueuedCompletionStatus) 함수로부터 완료 통지가 떨어지기 전까지 WSASend나 WSARecv를 다시 걸지 않는다. 보낼게 있는데 아직 완료통지가 떨어지지 않은 경우에는 SendQ에 enqueue만 하고 WorkerThread에서 완료 통지 후처리 작업이 끝났을 때 모여있는 패킷을 한꺼번에 WSASend하면 된다.

3.2.2 IO reference count

네트워크 엔진에서는 이상한 패킷을 보내지 않는다면(확인 코드가 맞지 않다거나, 존재하지 않은 패킷 번호거나, 디코딩 결과가 맞지 않거나) 서버쪽에서 먼저 끊는 일은 없다. 클라이언트 쪽에서 끊어질 경우 해당 세션의 소켓을 닫고 세션의 재사용을 위해 반환한다. 하지만 그냥 WSASend나 WSARecv호출 시 연결 끊김이 감지 되었다고 해서 리소스를 반환해버리면 문제가 돼 버린다. 예를 들면 WSASend에서 연결 끊김이 감지되어 세션과 소켓을 반환했는데 이 시점이

해당소켓으로 WSARecv에서 걸어논 IO작업 완료통지의 후처리를 IO WorkerThread에서 완료하기 전 이라면 문제가 생길 수 있다. GQCS에서는 세션 포인터를 키로 받는데 이 세션이 이미 내 세션이 아닐 수가 있다. 내 세션이 아닌데 건들여서는 안 된다. 그러므로 WSASend와 WSARecv를 걸기 전에 IO reference count를 증가하고 연결 실패감지나 완료통지때 차감하는 식으로 하여 해당 세션의 IO작업이 모두 완료되었다고 판단되면 그때 리소스를 반환한다.

3.2.3 Session Management

세션들은 동기화 대상이다. 특정 세션을 컨테이너에서 찾아 작업을 하려 하는데 이미 삭제되어 있을 수도 있다. 그래서 삽입, 삭제가 가능한 컨테이너에서 관리하게 될 경우 세션 컨테이너 Lock과 세션 Lock을 중첩해서 걸어야 한다. 하지만 이 방식은 구조가 더럽기 때문에 세션 배열을 사용하여 삽입, 삭제가 아닌 사용, 미사용을 표기하여 재사용 구조로 간다면 Lock을 중첩해서 걸지 않고 세션 배열 Lock과 세션 Lock을 분리할 수 있다. 해당 엔진에서는 방금 말한 배열 구조에 더해 동기화 객체를 사용하여 동기화를 잡는 방식이 아닌 IO reference count와 release flag 변수들로 InterLock을 사용하여 동기화를 잡았다.

세션 아이디는 8바이트 자료형에 저장되는데 이 중 6바이트만 식별자로 사용하고 뒤의 2바이트는 배열 인덱스를 저장하여 세션 아이디를 통해 쉽게 세션 객체에 접근할 수 있도록 했다.

3.3 test

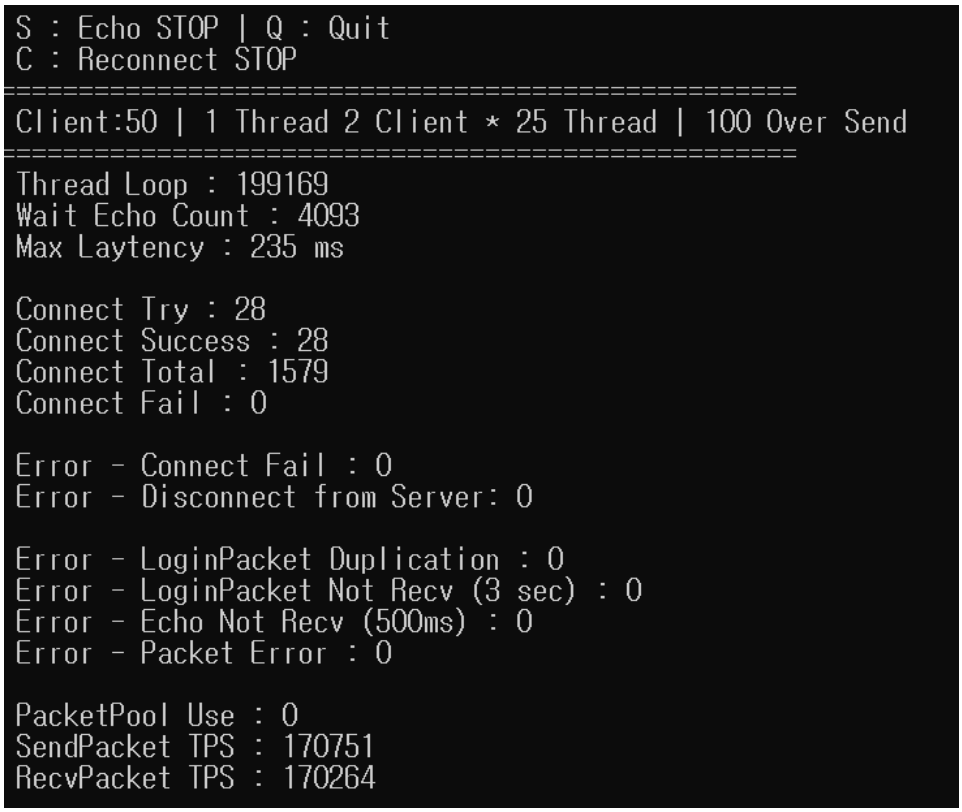

그림 6 Echo Dummy

만들어진 엔진에 에코기능만 있는 컨텐츠를 구현 한 후, 더미 클라이언트를 통해 문제 없이 돌아가는지 테스트 했다. 더미 클라이언트에서는 사용자가 지정

한 만큼의 연결을 한다. 연결 후 연결이 실패하지는 않았는지, 또 보낸 에코메세지를 제대로 받고 있는지(보낼 때 수치를 증가하고 받으면 수치를 차감한다. s를 누르면 보내기를 중단하며 이때 수치는 0이 떠야 한다) 또 서버에서 임의로 끊어버리는 것은 없는지 확인한다. 추가로 옵션을 주면 단순 연결+에코메세지 테스트 뿐만 아니라 연결을 끊고 재연결 테스트까지 하는 기능을 사용해 엔진쪽에서 오류가 나지 않는지(엉뚱한 세션에 이상한 메시지가 오지는 않는지) 확인했다.

3.4 Chatting Server

자신의 섹터에만 broadcasting을 하는 채팅서버를 구현하고자 한다면 (for mmorpg) 위 엔진을 사용하여 어떻게 구현할 수 있을까? 크게 두가지 버전이 있다. 첫째는 위에서 말한대로 엔진의 의도에 맞게 OnRecv에 패킷 프로시저, 마샬링과, 컨텐츠 파트를 넣어 IOCP WorkerThread에서 모든 일을 처리하는 방법이다. 공유자원이 있기는 하지만 단순한 컨텐츠기 때문에 공유자원을 분리하여 동기화 객체를 사용하는 것이 어렵지 않다. 하지만 해당 구조는 로그인 서버에서 구현할 것이기 때문에 채팅 서버는 조금 다른 구조로 가겠다.

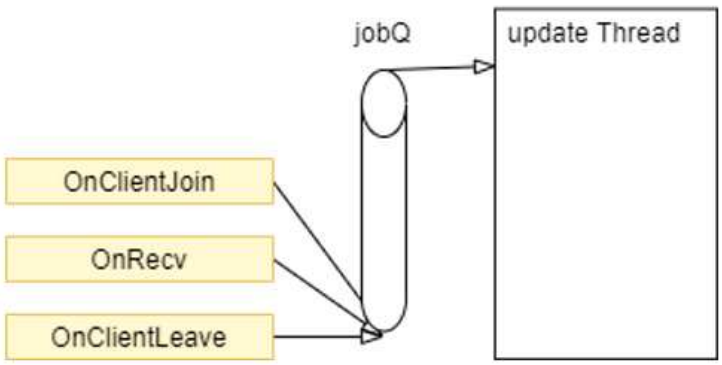

그림 7 Chatting Server

채팅 서버에서 하는 일은 유저가 섹터 이동 요청시 섹터 이동, 메시지 보내기 요청시 해당 섹터에 메시지 broadcasting 하는 것이 전부다. 하지만 유저와 섹터(플레이어 리스트를 들고 있다)를 STL 컨테이너에서 관리할 경우 스레드에 안전하지 않기 때문에 여러 스레드에서 동시에 컨테이너에 접근하여 작업을 할 경우(WorkerThread에서 호출되는 OnRecv에서 컨텐츠 작업을 병렬적으로 진행할 경우) 동기화에 문제가 생길 수 있다. 하지만 위 그림과 같이 유저 생성(OnClientJoin) 패킷 처리(OnRecv) 유저 삭제(OnClientLeave)를 WorkerThread에서 병렬적으로 진행하는 것이 아닌 하나의 큐를 통해 단일 스레드에서 직렬적으로 진행 한다면 동기화가 필요하지 않다. 큐는 2장에서 만든 스레드에 안전한 LockFreeQueue를 사용한다. 테스트는 채팅 더미 클라이언트(구조는 위에서 말한 에코더미 테스트와 유사)를 이용하여 문제없이 컨텐츠가 돌아가는지 확인했다.

3.5 Login Server

로그인 서버를 만드는 이유는 게임 서버의 부담을 조금이라도 줄이기 위함이다. 사용자가 타 플렛폼으로부터 토큰을 받아오고 우리가 그 토큰으로 해당 플렛폼으로부터 인증을 받는 구조라면 이 무거운 행위를 게임 서버로부터 분리하기 위해 로그인 서버가 존재하는 것이다. 로그인 서버가 하는 일은 다음과 같다.

1. 사용자가 가져온 토큰으로 타 플랫폼으로부터 인증을 거침
2. 인증이 완료가 되면 해당 토큰을 게임 서버와 공유하는 db에 저장

같은 상황을 만들기 위해 1번의 경우에는 타 플랫폼대신 db로부터 인증을 거치는 과정으로 대체했고 2번의 공유 db는 memoryDB인 redis를 사용했다.

구현해야 할 기능들은 각 세션별 독립적으로 수행되고 공유하는 자원도 없어 동기화도 필요 없기 때문에 3장에서 만든 네트워크 엔진에서 구현하기에 가장 알맞은 형태의 서비스다. 두 작업 모두 굉장히 무거운 작업이기 때문에 WorkerThread 개수와 실제 돌아하는 WorkerThread의 개수를 잘 조절해야 한다. 실제 돌아가는 WorkerThread의 개수는 다른 스레드들과 코어 개수에 맞게 정해져야 하고 WorkerThread의 개수는 위와 같이 IO 작업이 많으면 많을수록 더 큰 값으로 정해저야 한다. (그래야 IO 작업을 위해 Block을 당해도 다른 놀고 있는 스레드들이 실제 돌아가는 WorkerThread 개수를 맞춰 추가로 돌아간다)

3.6 Chatting Server with Login Server

위에서 만든 채팅 서버에서 위에서 만든 로그인 서버로부터의 인증과정을 거치게끔 한다면 전체적인 구조는 다음과 같다.

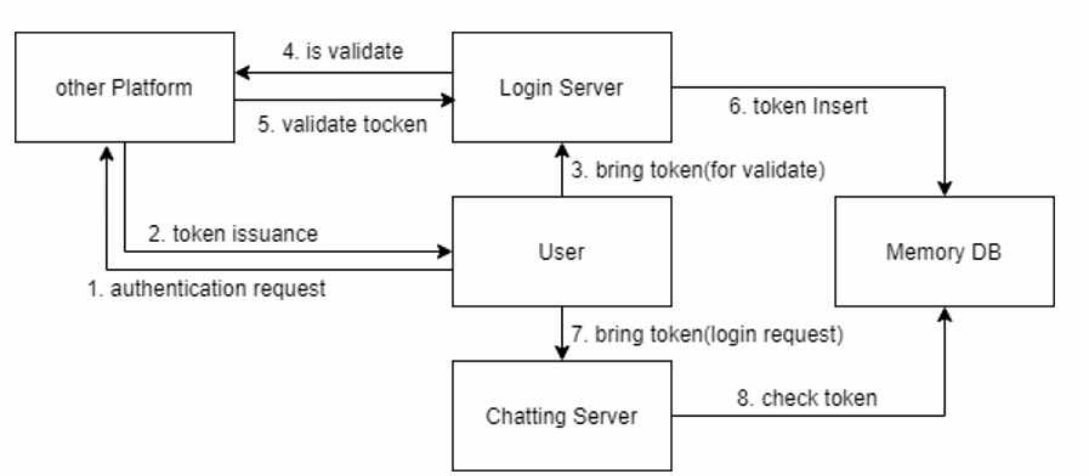

그림 8 Chatting Server with Login Server

단, 주의해야 하는 것은 Login Server의 경우에는 무거운 작업들을

WorkerThread에서 직접 수행하고 WorkerThread는 여러개가 병렬적으로 진행되기 때문에 문제 없지만, Chatting Server에서는 동기화를 위해 UpdateThread 하나에서 작업을 직렬적으로 처리하고 있으므로 만약 여기서 그냥 redis에 토큰 확인작업이 들어간다면 UpdateThread의 TPS가 현저하게 떨어진다. 그래서 해당 파트는 IOCP를 사용하여 다른스레드에게 비동기적으로 작업을 던지고 작업이 완료되면 jobQ에 완료통지를 던져 UpdateThread에서 후처리를 하게 한다. (유효한 토큰이라면 인증완료 패킷 발송 유효하지 않은 토큰이라면 실패 패킷 발송)

제 4 장 Network Engine for Game

4.1 overview

3장에서 만든 엔진을 기반으로 게임 서버를 구현한다면 어떠한 구조로 짜야할까? 첫째로는 3장에서 구현한 로그인 서버와 같이 WorkerThread 에서 컨텐츠 파트 까지 다 수행하게끔 짜는 방법이 있다. 하지만 대부분의 게임의 경우 공유자원이 있고 또 이 자원이 Lock을 걸기 깔끔하게 나누어져있지 않다. 그럼 두번째 방법으로 채팅 서버와 같은 방식으로 짜는 것은 어떨까? jobQ를 중간에 두고 UpdateThread는 WorkerThread들로부터 작업을 직렬적으로 받아 동기화 걱정없이 처리하는 방식이다. 하지만 이 경우 큐에 대한 부담이 너무 크다. 이와 같이 3장의 엔진으로는 동기화 문제가 복잡한 게임 서버의 성능을 뽑아내기가 힘들다. 4장에서는 3장에서 만든 엔진을 조금 변형시켜 게임을 위한 엔진을 구현했다.

4.2 structure

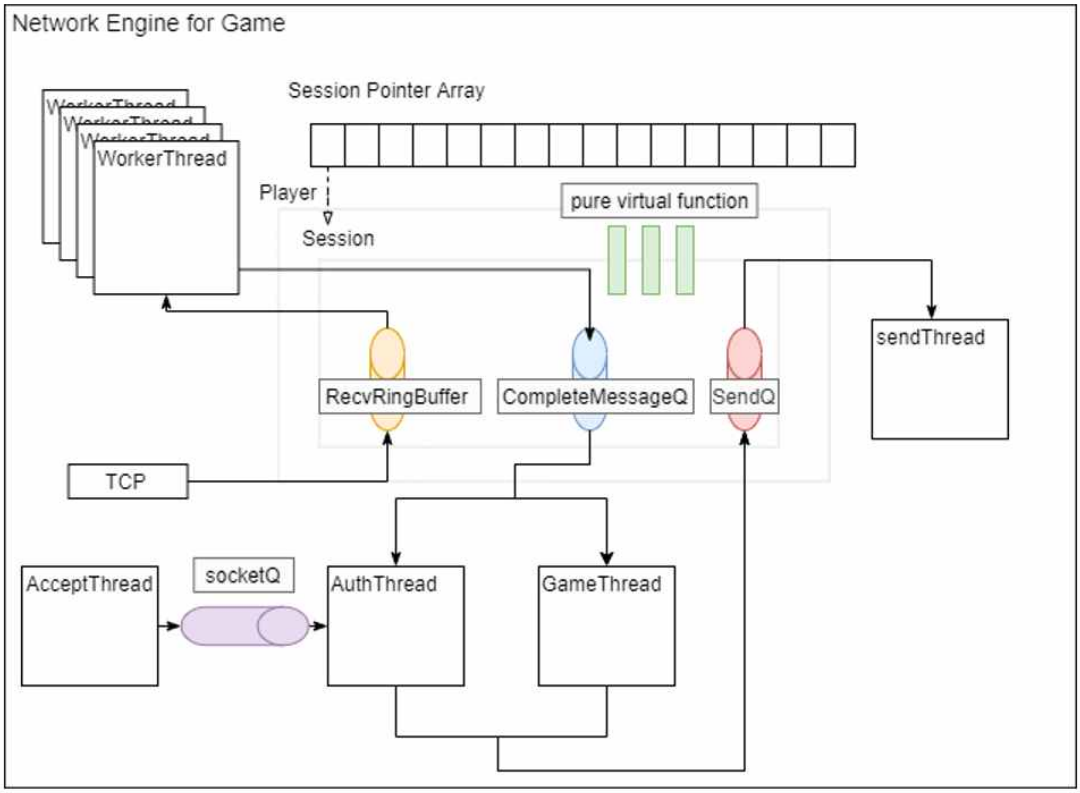

그림 9 Network Engine for Game

엔진 사용자가 해야 할 것은 두 가지다. 첫 번째는 세션을 상속받은 플레이어를 구현해야 한다. (순수 가상함수) 구현 후 플레이어 객체를 희망 최대 동접자 수 만큼 만들어 엔진의 함수를 호출하면 내가 만든 객체가 세션 배열 포인터에 셋팅된다. 두 번째는 네트워크 엔진을 상속받은 게임 네트워크 엔진을 구현해야 한다.(순수 가상함수) 이 두 가지를 완료한 후에 네트워크 엔진 객체를 참조하여 Start를 호출해주면 네트워크 엔진 내부에서는 스레드가 돌면서 우리가 구현한 가상함수를 호출시켜주게 된다.

엔진 내부의 전체적인 흐름은 다음과 같다. GQCS에서 완료통지가 떨어져 RecvRingBuffer에 데이터가 들어오게 되면 WorkerThread에는 이를 하나의 메시지 단위로 만들어 패킷을 생성한다. 3장의 네트워크 엔진에서는 만들어진 패킷을 인자로 우리가 구현한 OnRecv 순수가상함수를 호출시켜주었는데, 여기서 하는 일은 그냥 CompleteMessageQ에 삽입하는 것이 전부다. AuthThread와 GameThread에서는 전체 세션을 순회하며 CompleteMessageQ에 있는 패킷들을 처리한다. 그리고 보낼 것이 있다면 패킷을 만들어 SendQ에 enqueue한다. sendThread에서는 세션을 순회하며 SendQ에 패킷이 있다면 보낸다. AuthThread, GmaeThread, SendThread는 전체 세션을 순회하며 사용자 지정 프레임 단위로 돌기 때문에 반응성과 cpu 사용량의 관계를 고려하여 프레임을 지정하면 된다.

4.2.1 AuthThread

AuthThread를 따로 둔 이유는 GameThread에 인증과정과 유저의 데이터를 db로부터 모두 긁어오는 과정을 넣기에 너무 부담스럽기 때문이다. 구체적으로 하는 일은 다음과 같다.

1. socketQ로부터 신규 접속자 처리
2. AuthThread에 입장한 세션의 후처리를 위해 Auth 입장 함수 (Session’s pure virtual function) 호출
2. 세션전체를 순회하며 아래과정 진행
3. Auth 패킷 프로시저(Session’s pure virtual function) 호출
4. Auth update함수(Session’s pure virtual function) 호출
5. Auth에서의 작업이 끝나 Game으로 넘어가길 원하는 세션들의 상태 변경
6. 연결이 끊긴 세션들에 대한 처리

7. AuthThread를 떠나는 세션의 후처리를 위해 Auth 퇴장 함수 (Session's pure virtual function) 호출

4.2.2 GameThread

실질적인 게임 컨텐츠가 진행된다. 하는 일은 다음과 같다.

1. 세션전체를 순회하며 아래과정 진행
2. Auth에서 Game으로 오길 원하는 세션에 대해 신규 접속자 처리
3. GameThread에 입장한 세션의 후처리를 위해 Auth 입장 함수 (Session's pure virtual function) 호출
4. Game 패킷 프로시저(Session's pure virtual function) 호출
5. Game update함수 (Session's pure virtual function) 호출
6. 연결이 끊긴 세션들에 대한 처리
7. GameThread를 떠나는 세션의 후처리를 위해 Game 퇴장 함수 (Session's pure virtual function) 호출

4.3 test

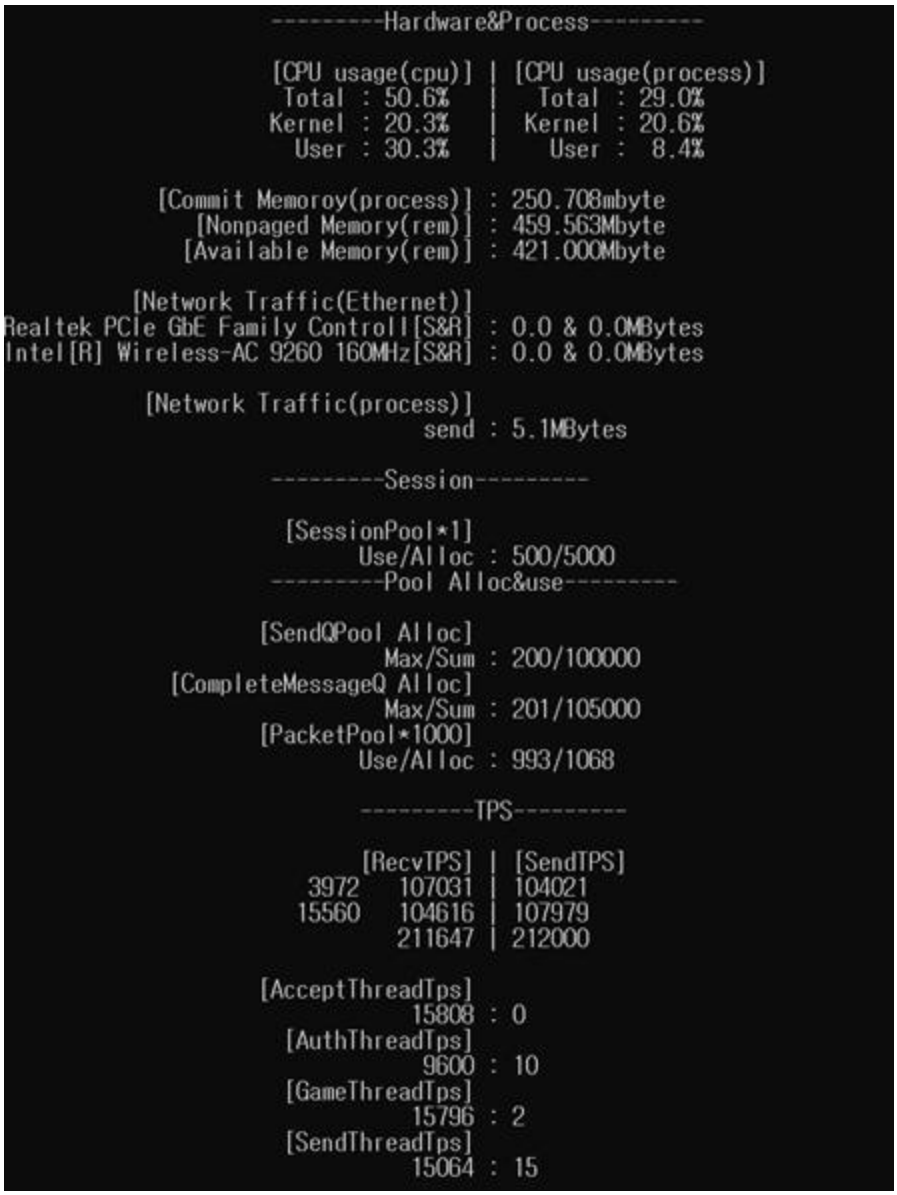

그림 10 Monitoring

3장에서처럼 에코테스트를 진행하여 로직상 문제가 없는지 확인했고 pdh를 통해 윈도우에서 수집하고 있는 모니터링 요소들을 얻어와 확인했다.(cpu 사용량, 네트워크 트래픽, 메모리 등) 또 MemoryPool을 이용하고 있는 모든 부분을 수치로 표기하여 메모리 누수는 없는지 확인할 수 있었다. 성능은 WorkerThread의 send recv TPS와 AuthThread, GameThread, SendThread의 프레임이 몇이 나오는지 확인했다.

4.4 structural limitations

해당 엔진의 특수한 점은 AuthThread와 GameThread에서는 각기 컨테이너를 두어 해당하는 세션만 관리하는 것이 아닌 세션 배열의 전체(비어있는 세션 포함)를 순회하며 상태값에 따라 나누어 세션 처리를 하고 있다는 점이다. 이러한 구조로 짠 이유는 동접이 적다면 전체 세션을 순회하는 것이 의미 없는 일이지만, 동접이 꽉 차 있다면 반대로 컨테이너에서 관리되고 있는 세션을 이터레이터로 순회 하는 것과 스레드간의 객체이동을 위해 enqueue, dequeue 작업을 하는 것이 더 의미 없는 행동이 된다. 동접이 모두 찼을때의 성능을 위해 설계된 엔진이다. 단, 이 구조는 몇 가지 단점을 가지고 있다. 첫째는 확장이 힘들다. 두번째는 많은 코어를 알맞게 활용하는 방식이 아니다. 게임의 복잡한 공유자원에 대한 동기화 작업으로부터 자유롭기 위해 GameThread를 단일 스레드로 갔기 때문에 여기서 성능을 올리려면 GameThread 개수를 늘리는 방식이 아닌, GameThread 내부의 무거운 작업을 비동기로 빼는 정도로 만족해야 한다.

제 5 장 결론

다양한 서비스에 이용될 수 있는 두 가지 버전의 네트워크 엔진을 만들었다. 하나는 로그인 서버와 채팅서버를 위한 네트워크 엔진, 다른 하나는 이를 개량하여 게임서버를 위한 네트워크 엔진을 만들었다. 이렇게 분리해서 만들어야 했던 이유는 게임의 경우 공유자원 때문에 세션들이 독립적으로 컨텐츠를 진행하기 힘든 구조고 그렇다고 동기화 없이 직렬적으로 처리되기 위해 중간에 큐를 하나 두어 사용하기에는 큐의 부담이 너무 컸다. 그래서 세션이 각기 큐를 가지고 AuthThread와 GameThread 에서 세션 전체를 순회하며 큐로부터 패킷을 뽑아 처리하는 방식으로 큐에 대한 부담을 지웠다. 단, 이 엔진에도 한계점이 있는데 확장성을 고려한 엔진은 아니며, 하드웨어의 성능이 우수한 환경에서 코어의 성능을 모두 뽑아 사용할 수 있는 구조는 아니다.