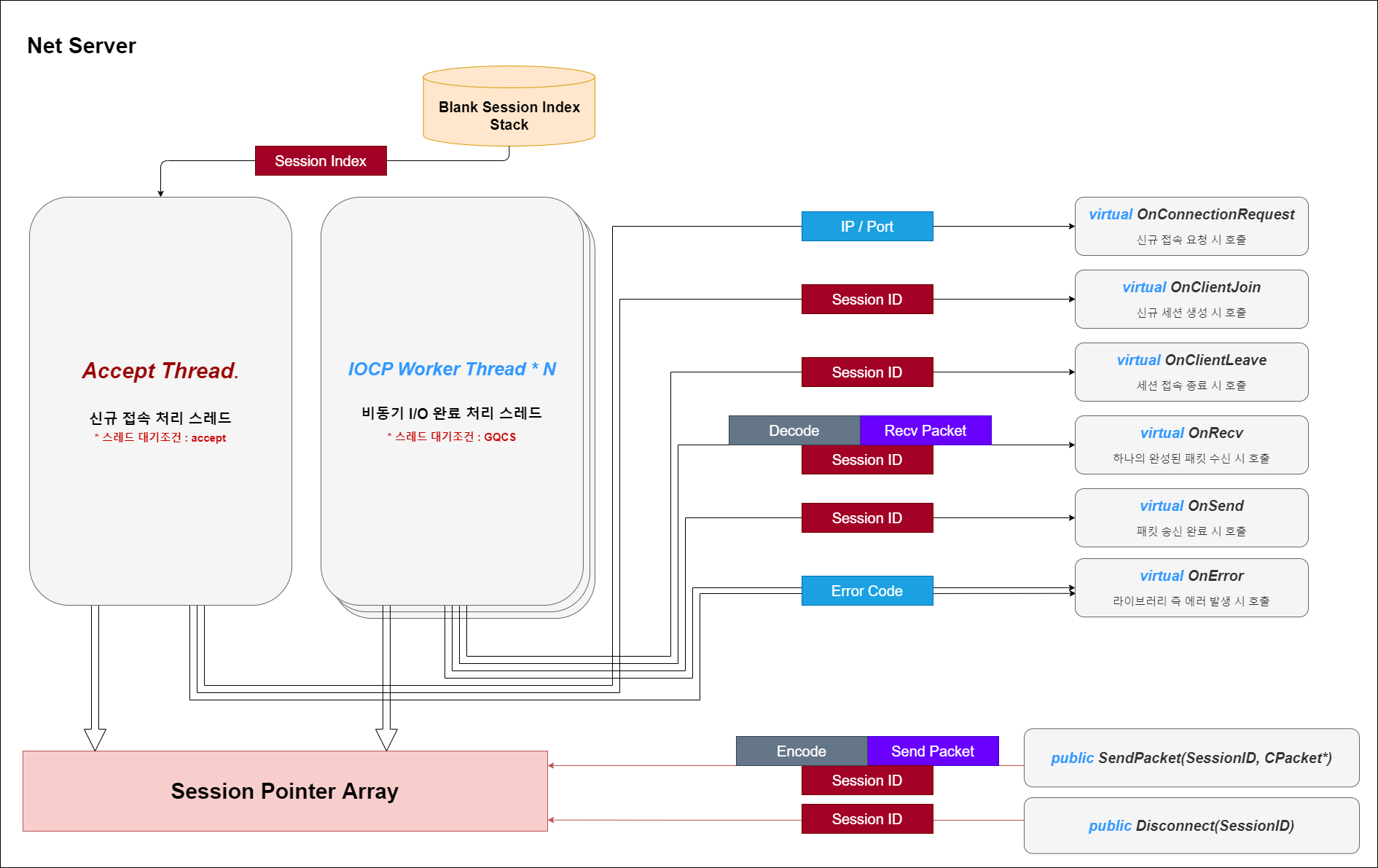
**목차**

1. **Network Library** 
   * **Chat Server**
   * **Login Server**
   * **Monitor Server**
2. **MemoryPool TLS**
3. **Lock Free Queue**

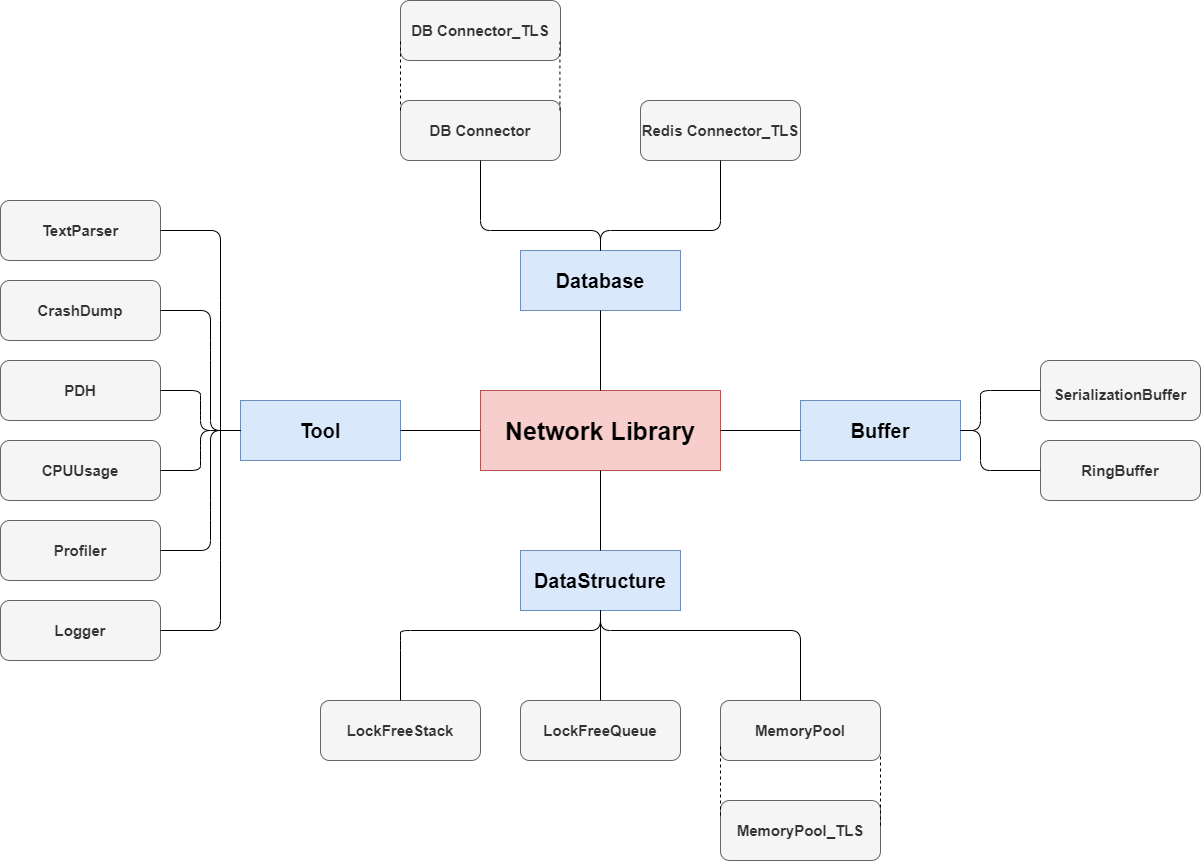
**사용 기술**

* + **C/C++**
  + **IOCP**
  + **MySQL**
  + **Redis**

**1. Network Library**

****

1. **구성도**

****

1. **설계 목적**
   * 서버의 Contents 담당 계층과 Network I/O 담당 계층을 구분하기 위함.
   * IOCP를 사용하여 비동기 Network I/O 완료 처리를 멀티스레드로 처리하기 위함.
2. **패킷 프로토콜 구조**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

1. **패킷 구조**
   * Header 영역은 Encode에서만 사용. 실제 사용할 수 있는 영역은 WritePos 지점부터 사용할 수 있다.

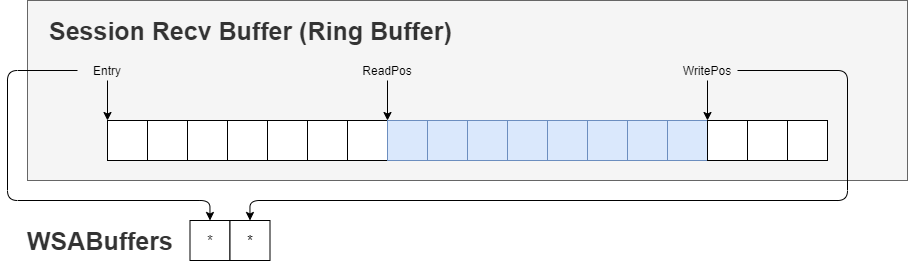
텍스트, 모니터, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

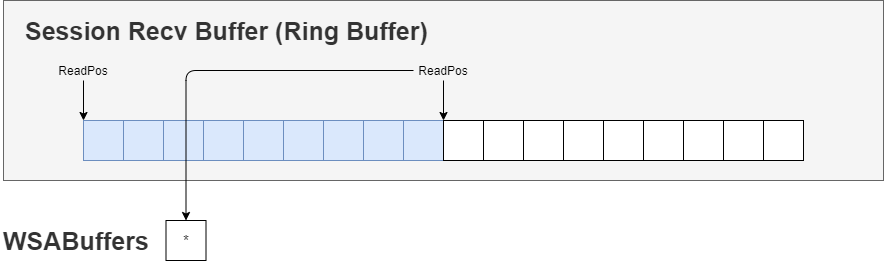
1. **기본 제공 함수**

* **bool Start(BindIP, BindPort, IOCPWorkerThread, ActiveWorkerThread, bNagle, bZeroCopy)**
  + **라이브러리 가동 시 사용.**
  + **매개변수로 라이브러리 구성 정보 설정**
* **void Stop(Type)**
  + **라이브러리 정지 시 사용**
* **void Restart(void)**
  + **정지 상태의 라이브러리 재가동 시 사용**
* **bool SendPacket(*SessionID, \*Packet*)**
  + **패킷 송신 요청 시 사용**
* **bool Disconnect(*SessionID*)**
  + **세션 연결 종료 요청 시 사용**

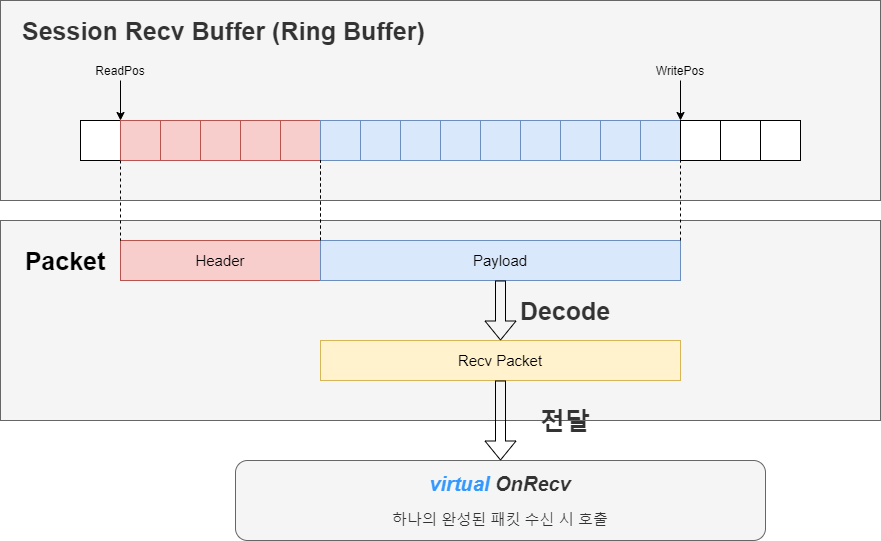
1. **송/수신 처리 방식**
2. **수신 등록 (RecvPost)**
   * 수신 버퍼에 남은 공간이 아래와 같은 경우, WritePos와 Entry가 수신 버퍼로 등록된다.

****

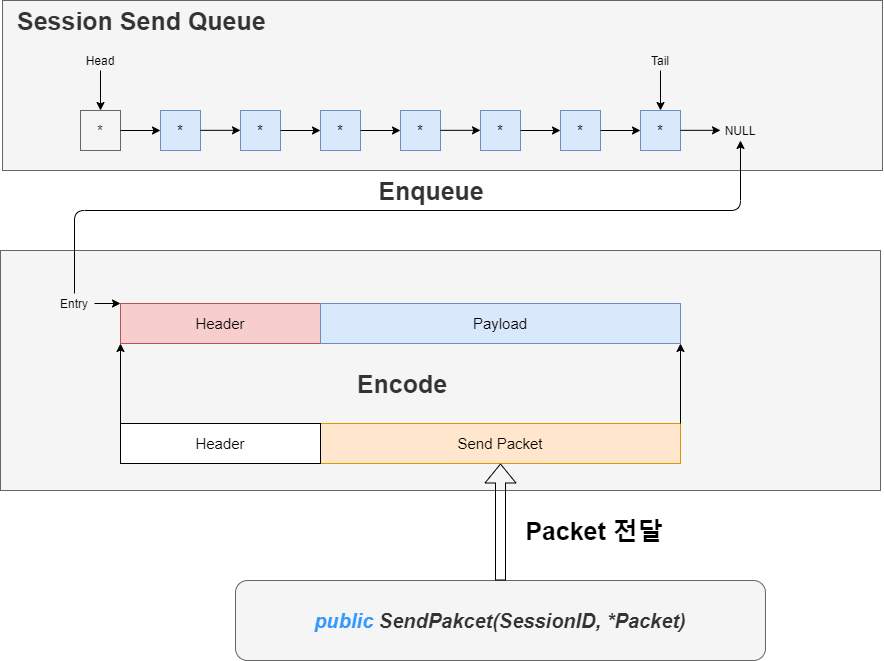
* + 수신 버퍼에 남은 공간이 아래와 같은 경우, WritePos만 수신 버퍼로 등록된다.

****

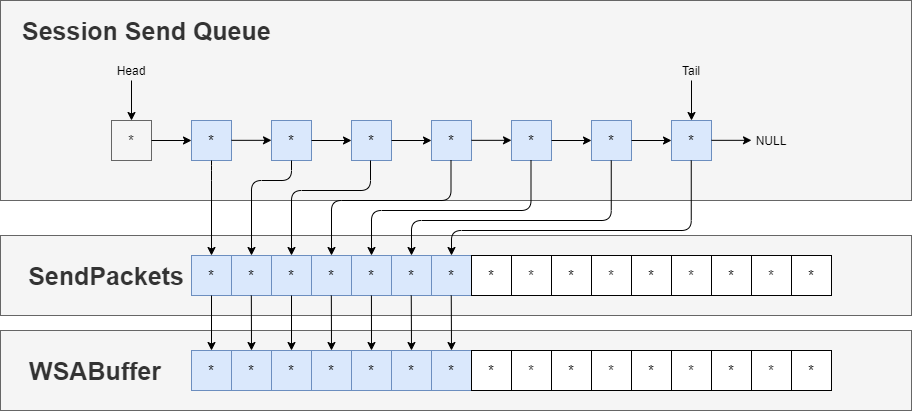
1. **수신 패킷 처리 (RecvCompletedProcedure)**
   * RecvPost()에서 등록한 세션 수신 버퍼에 데이터가 존재할 때 IOCPWorkerThread에서 실행된다.
   * 수신 버퍼에 패킷이 완성된 경우, 복호화 후 콘텐츠 계층에 패킷을 전달한다.
   * 수신 버퍼에 더 이상 처리할 데이터가 없는 경우(헤더 또는 패킷 크기 미만의 데이터만 존재) 해당 로직을 종료한다.



1. **패킷 송신 (SendPacket)**
   * 컨텐츠 계층으로부터 전달 받은 패킷을 내부적으로 암호화 후 대상 세션의 송신 버퍼에 추가한다.
   * 이때, 암호화 과정에서 컨텐츠 계층 패킷에 라이브러리 계층 헤더가 추가된다.

****

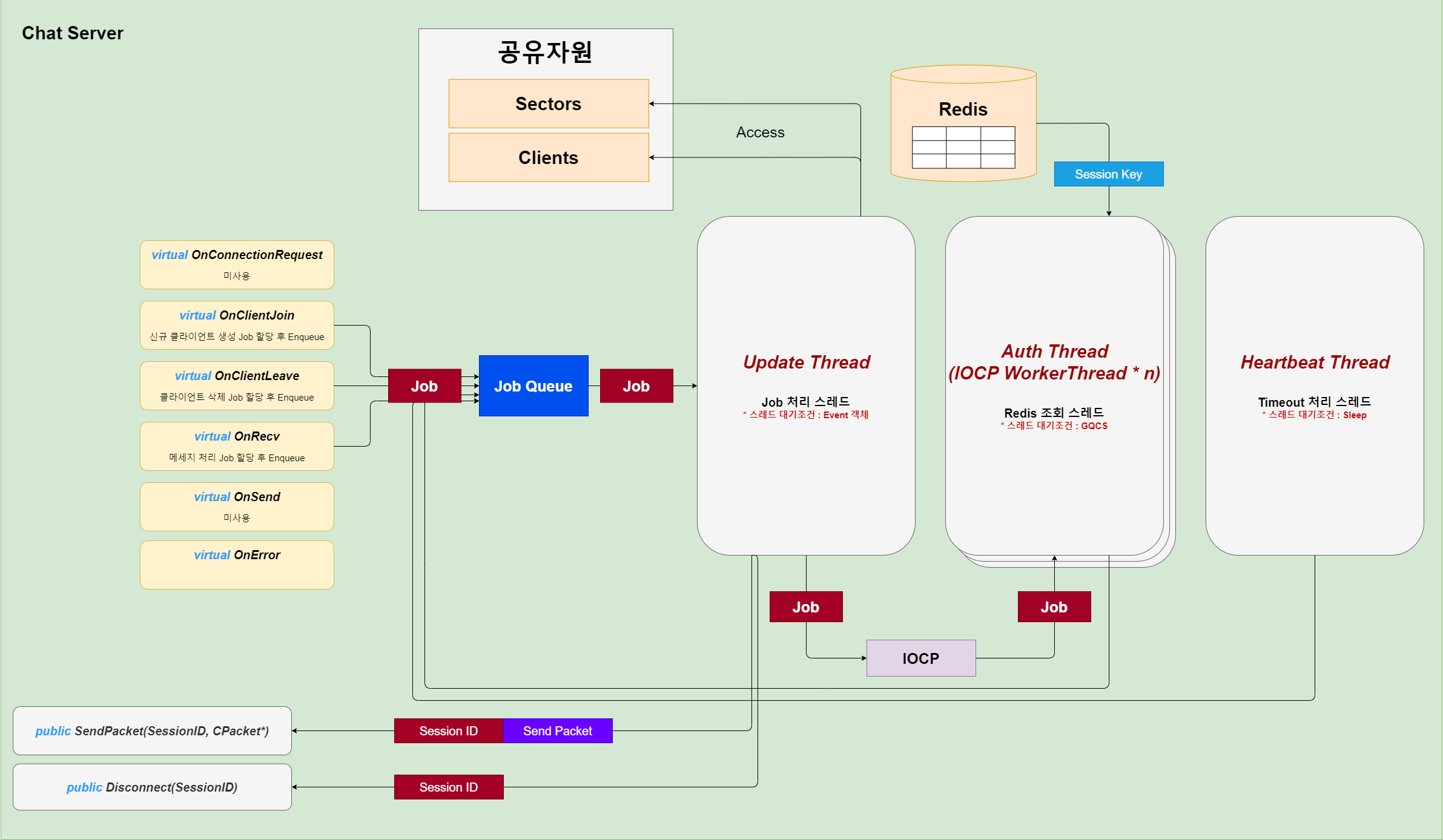
1. **송신 등록 (SendPost)**
   * 송신 완료 처리 부분에서 패킷 반환을 위해 WSABuffer에 등록할 패킷에 대한 포인터를 배열에 등록.
   * 세션 SendQueue에 존재하는 노드 개수만큼 WSABuffer에 등록.

****

1. **송신 패킷 처리 (SendCompletedProcedure)**
   * SendPackets에 등록된 패킷에 대한 래퍼런스 카운트를 차감한다.
   * SendFlag를 갱신한 후 다시 SendPost 호출을 통해 송신을 등록한다.
2. **검증 테스트**

|  |  |  |  |
| --- | --- | --- | --- |
| **테스트 시나리오** | | | |
| **서버명** | Echo Server | 시작일 | 2021.11.08 |
| **테스트 대상** | 부하테스트 (세션할당/해제, 송/수신) | 종료일 | 2021.11.15 |
| **테스트 결과** | 이상 없음. 평균 RecvPacket TPS : 70만 / SendPacket TPS : 80만 | | |
| **테스트 사양** | Server : 4(4) Core  Dummy : 2(2) Core | | |
| **번호** | **테스트 케이스 및 예상 결과** | | |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | 클라이언트 접속 요청  접속 처리 확인  패킷 송신  송신 패킷과 동일한 패킷을 수신   * + 500ms 안에 송신 패킷을 수신 받지 못한 경우 **Echo Not Recv** 카운팅   + 잘못된 패킷을 수신 받은 경우 **Packet Error** 카운팅   다음 패킷 송신  연결 종료 요청   * + 요청 전에 서버가 먼저 끊어버린 경우 **Disconnect from Server** 카운팅   연결 종료 확인 | | |

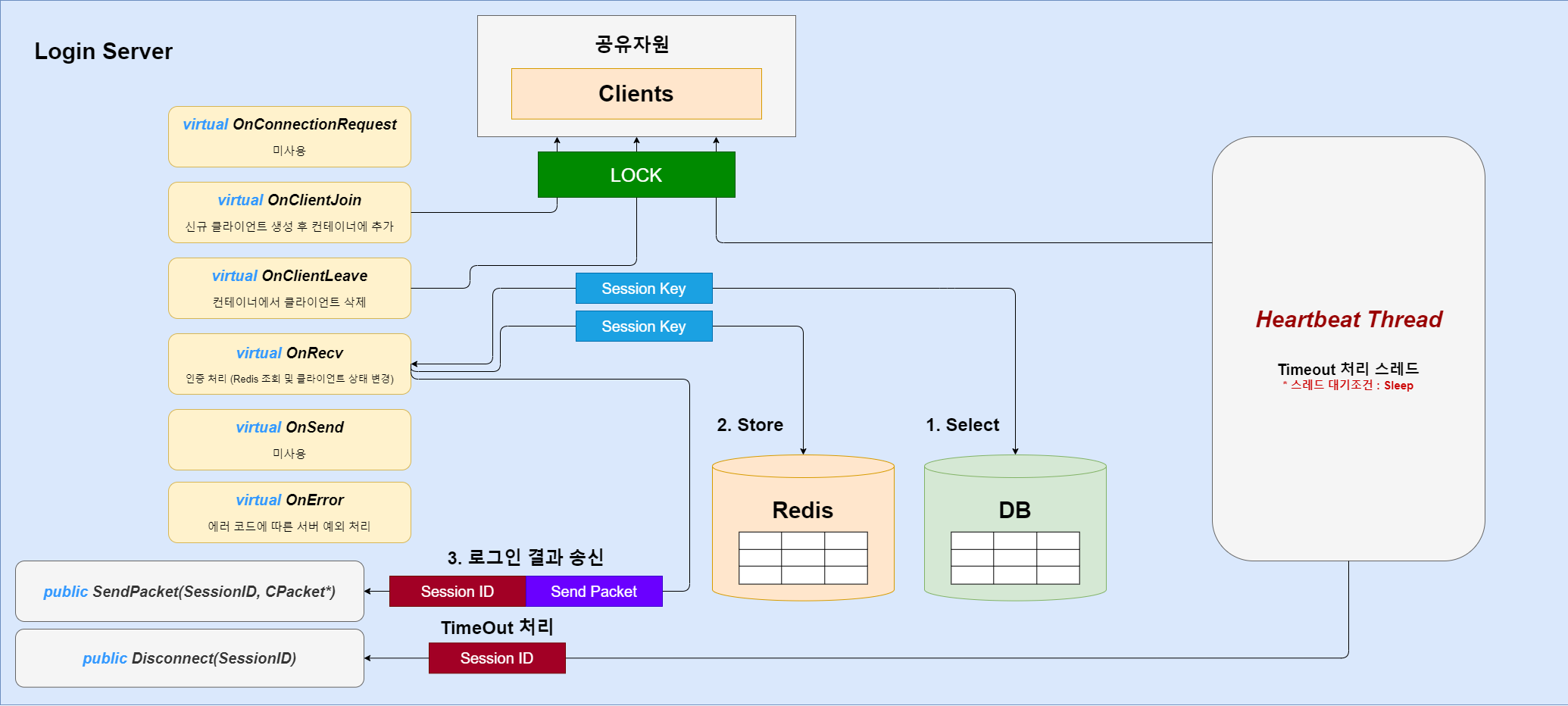
**Chat Server**

****

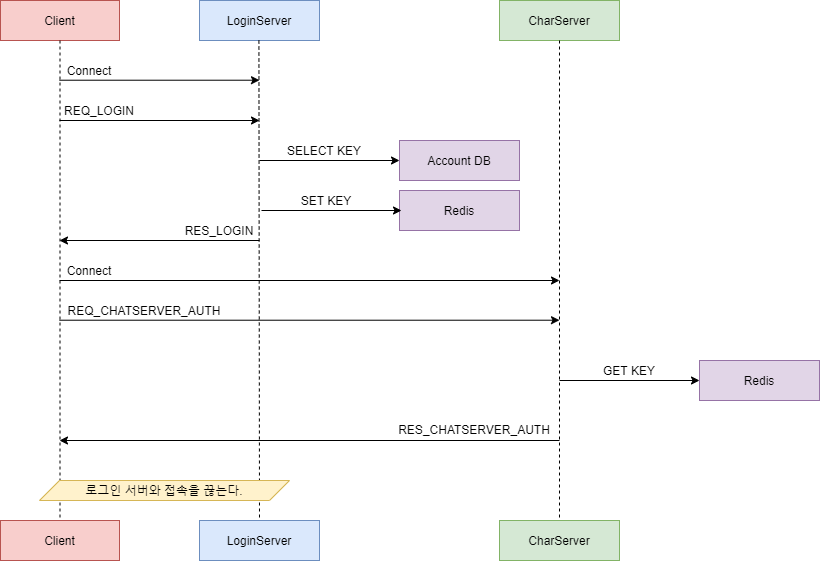
1. **설계 목적**
   * 모든 네트워크 이벤트 처리 로직을 싱글스레드 기반에서 처리한다.
   * 공유 자원에 대한 스레드 동기화를 신경쓰지 않게끔 한다.
   * 공유 자원에 대한 스레드 동기화에 대한 성능 저하를 개선한다.
   * 로그인 인증 처리를 위한 Redis 접근 시 별도의 IOCP Worker Thread를 사용하여, Update Thread의 성능을 향상시킨다.
2. **본 설계의 단점**
   * 모든 로직 처리를 Update Thread에서 처리하고 있으므로 병렬성 및 응답성이 떨어진다.
   * 모든 Job을 하나의 Lock-Free Queue에 Enqueue하므로, 스레드 간 경합이 빈번하다.
3. **검증 테스트**

|  |  |  |  |
| --- | --- | --- | --- |
| **테스트 시나리오** | | | |
| **서버명** | Chat Server | 시작일 | 2021.11.15 |
| **테스트 대상** | 부하테스트 (로그인 처리 및 에러 검증) | 종료일 | 2021.11.22 |
| **테스트 결과** | 이상 없음. Update Thread TPS : 14000 ~ 16000 | | |
| **테스트 사양** | Server : 4(4) Core  Dummy : 2(2) Core | | |
| **번호** | **테스트 케이스 및 예상 결과** | | |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | 클라이언트 접속 요청  접속 처리 확인  요청 패킷 송신 (로그인 요청, 섹터 이동, 채팅 송신)  요송 패킷 결과에 맞는 패킷 수신   * + 요청 송신 패킷에 대한 결과를 수신 받지 못한 경우 **ReplyWait 카운팅**   다음 요청 패킷 송신  연결 종료 요청   * + 요청 전에 서버가 먼저 끊어버린 경우 **DownClient** 카운팅   연결 종료 확인 | | |

**Login Server**



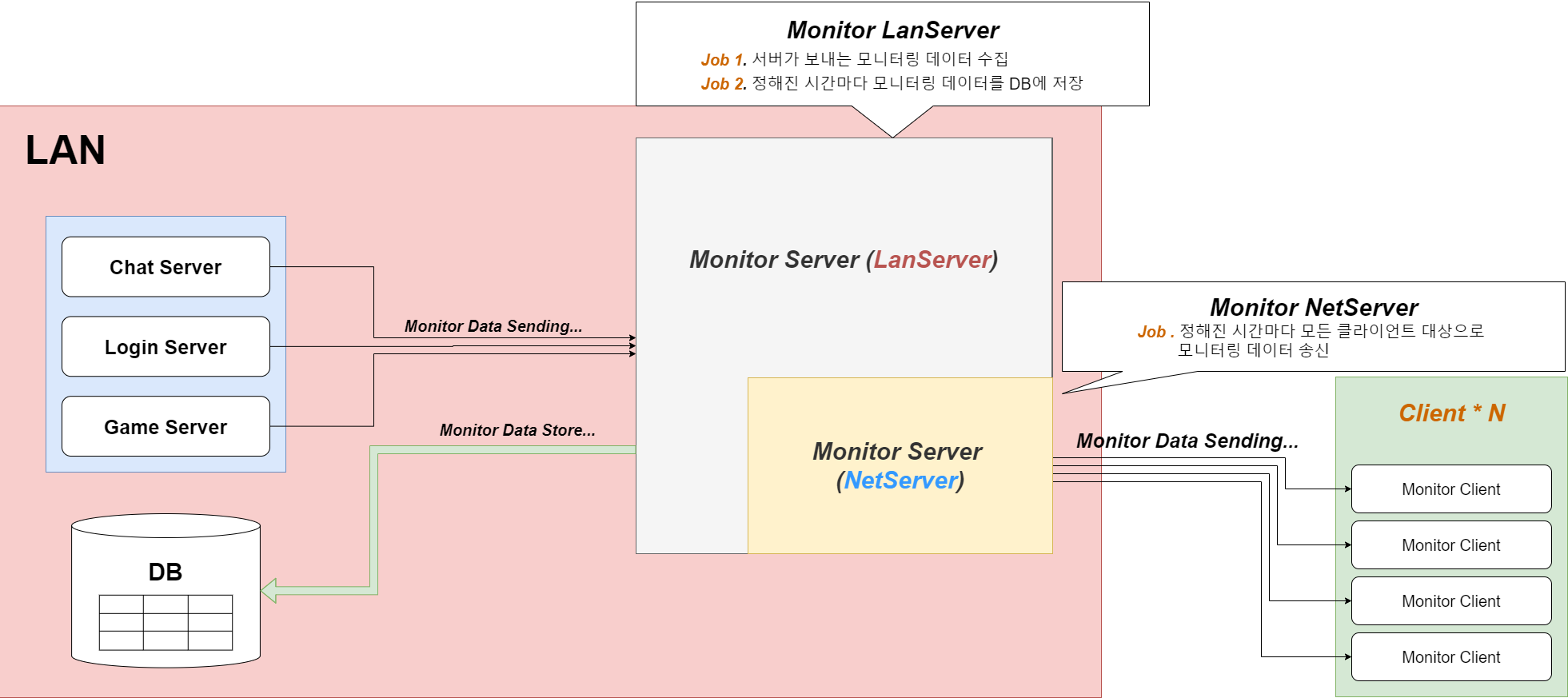
1. **설계 목적**
   * 모든 네트워크 이벤트 처리 로직을 IOCP Worker Thread에서 처리한다.
   * 로그인 인증 처리를 위한 DB 접근 과정을 IOCP Worker Thread에서 세션마다 병렬로 처리한다.
   * IOCP의 기능을 사용하여 DB 접근 시 스레드 대기 상태 동안 다른 스레드를 Active 시킴으로 서버의 병렬성을 상승시킨다.
2. **로그인 처리 과정**

****

1. **검증 테스트**

|  |  |  |  |
| --- | --- | --- | --- |
| **테스트 시나리오** | | | |
| **서버명** | Login Server | 시작일 | 2021.11.22 |
| **테스트 대상** | 오류 검증 테스트 | 종료일 | 2021.11.29 |
| **테스트 결과** | 이상 없음. Auth Success TPS : 150 ~ 200 | | |
| **테스트 사양** | Server : 4(4) Core  Dummy : 2(2) Core | | |
| **번호** | **테스트 케이스 및 예상 결과** | | |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  8  9 | 클라이언트 접속 요청  접속 처리 확인  로그인서버에 패킷 송신 (로그인 요청)  채팅 서버 정보를 수신 받는다.  로그인 서버 연결 종료 요청   * + 요청 전에 서버가 먼저 끊어버린 경우 **DownClient** 카운팅   채팅 서버에 패킷 송신 (로그인 요청)  로그인 결과 수신  채팅 서버에 연결 종료 요청   * + 요청 전에 서버가 먼저 끊어버린 경우 **DownClient** 카운팅   채팅 서버 연결 종료 확인 | | |

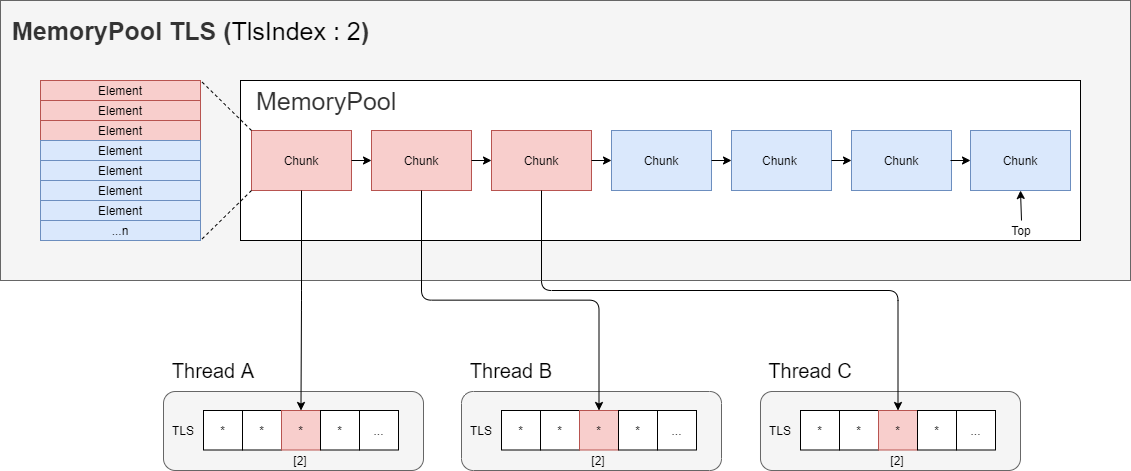
**Monitor Server**



1. **설계 목적**
   * 라이브러리 모듈 중 하나인 LanServer를 사용하여 서버 간의 통신 효율을 증대 시킨다. (패킷 하나 당 약 1.2ms 처리 시간 감소)
   * 외부 모니터링 클라이언트는 NetServer를 사용하여 통신한다.
   * 모든 모니터링 정보는 일정 시간 간격으로 DB에 Write 하여 백업한다.
     + DB 접근 스레드는 별도의 IOCP Worker Thread를 사용하여 로직 처리 효율을 상승시킨다.

**2. Memory Pool TLS**

1. **설계 목적 및 구조**
   * 현재 라이브러리에서 패킷 메모리 할당 및 해제는 하나의 **공용 메모리풀**을 대상으로 실행 중.
   * 때문에 Lock-Free Stack 구조이며, 스레드 간 경합이 발생할 경우 스레드 개수에 비례하여 성능 저하.
   * 위 이유로 TLS를 이용하여 할당에서의 스레드 경합을 최소화하기 위함.

****

1. **Chunk 구조**
   * 할당 및 반환을 위한 Chunk 구조는 아래와 같다.
   * Elements 사이즈가 클수록 경합 빈도는 낮아지지만, 메모리 사용량이 늘어난다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. **메모리 할당 절차**
   1. 메모리 할당 요청을 받았다.
   2. TLS에 세팅 된 Chunk가 없었다.
   3. MemoryPool에서 새로운 Chunk를 할당(경합 지점)받고 TLS에 세팅 후 Element를 반환해준다.
   4. 메모리 할당 재요청을 받았다.
   5. TLS에 세팅 되어있는 Chunk에서 다음 인덱스의 Element를 반환해준다.
2. **메모리 반환 절차**
   1. 메모리 해제 요청을 받았다.
   2. TLS에 세팅 된 Chunk의 FreeCount를 차감한다.
   3. 차감한 FreeCount가 MAX였다.
   4. 해당 Chunk를 MemoryPool에 반환한다. (경합 지점)
3. **성능 테스트**
   * Heap과 기본 메모리풀을 사용하였을 때, 스레드 개수에 비례하여 성능이 저하됨.
   * TLS를 사용한 메모리풀의 경우, Chunk 할당 이후부터는 스레드의 배타적 공간을 사용하므로 스레드 개수와 상관없이 동일한 속도를 보이며 성능이 향상됐다.

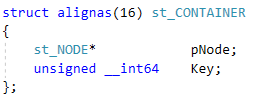
테이블이(가) 표시된 사진

자동 생성된 설명 **테이블이(가) 표시된 사진

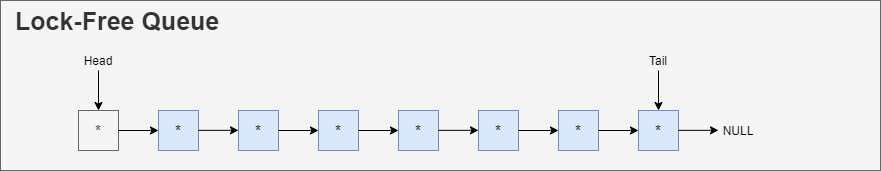
자동 생성된 설명**

**3. Lock Free Queue**

1. **구현 목적**
   1. 멀티 스레드 기반의 Lock-Free 라이브러리 구현을 위해 사용.
   2. 멀티 스레드 **디버깅 능력 및 이슈 추적 능력 증진**을 위해 구현.
2. **기본 구조**
   1. 노드의 추가, 삭제 및 Head, Tail의 변경을 위해 ***InterlockedCompareExchange/ InterlockedCompareExchange128*** 함수를 사용.
   2. 아래는 해당 함수 사용을 위한 컨테이너 구조체이며, Head, Tail의 메모리 구조다.



* 1. 각 스레드는 위 구조체를 사용하여 스레드마다 배타적으로 CAS, DCAS 함수의 인자를 준비한 후, 비교 및 변환을 원자적으로 실행
  2. 위의 이유로 해당 Lock-Free Queue의 구조는 아래와 같은 List 형식이다.

****

1. **함수 작동 방식**
   * **Enqueue**
     1. 메모리풀에서 신규 노드를 할당받는다.
     2. 현재 Tail의 NextNode를 할당 받은 노드로 갱신한다. (CAS)
     3. 할당 받은 노드를 현재 Tail로 갱신한다. (DCAS)
   * **Dequeue**
     1. 현재 Head의 NextNode에서 값을 출력한다.
     2. 현재 Head의 NextNode를 현재 Head로 갱신한다. (DCAS)
     3. 이전 Head를 메모리풀에 반환한다.
2. **검증테스트**

* **초깃값 세팅 후 아래 스레드마다 아래 절차 반복.**
  + 1. Dequeue 체크
    2. 초깃값 체크
    3. 변경 데이터 유지 체크
    4. 원복 데이터 유지 체크
    5. Enqueue 체크

1. **디버깅 환경**

|  |  |  |
| --- | --- | --- |
| **로그 구조체** | 테이블이(가) 표시된 사진  자동 생성된 설명텍스트이(가) 표시된 사진  자동 생성된 설명 | - 메모리상에서 확인하고자 하는 로그 명세 목록 |
| **로그 배열** |  | - 구조체 배열을 전역 또는 클래스의 프로퍼티로 선언하여 사용  - Interlock 함수로 스레드마다 접근할 배열 고유 인덱스 획득 |
| **로그 넘버링** | 텍스트이(가) 표시된 사진  자동 생성된 설명 | - 임의로 지정하는 코드상의 로직 파악을 위한 상수 |
| **로그 기록** |  | - 확인하고자 하는 당시 상황의 값을 매개변수로 하여 **SetLog()** 함수를 통해 로그 기록. |

1. **버그리포트 1 : Dequeue에서 Head Next가 NULL인 상황**
   1. **로그 디버깅**

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 테이블이(가) 표시된 사진  자동 생성된 설명 | **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** |
| 1. pHeadNext : NULL  2. Capacity : 1  3. TailNextNode : TailNode | 1. Head과 Tail이 동일 노드 참조  2. (1)의 상황임에도 Capacity는 1  3. TailNextNode가 NULL이 아님 | 1. Capacity가 1이므로 Dequeue 실행  2. Head를 HeadNextNode로 갱신(Tail 추월)  3. 이전 HeadNode 반환 (Tail 반환) |
| 4 | **5** | **6** |
| 테이블이(가) 표시된 사진  자동 생성된 설명 | 테이블이(가) 표시된 사진  자동 생성된 설명 | 테이블이(가) 표시된 사진  자동 생성된 설명 |
| 1. Enqueue에서 반환했던 Head(Tail)Node 할당  2. AllocNode Next 초기화 | 1. SnapTail 생성  2. SnapTailNextNode를 AllocNode로 갱신  3. AllocNode와 TailNode는 동일한 상황  4. 이로 인해 TailNextNode는 TailNode | 1. TailNextNode에 AllocNode 추가했음으로 Capacity 증가 (0 -> 1)  2. 하지만 실제 HeadNextNode는 NULL |

* 1. **시나리오**

|  |  |
| --- | --- |
|  | 1. Head와 Tail Node가 동일한 상황  2. Capacity는 1  3. TailNextNode는 NULL이 아님. |
|  | 1. 다른 스레드의 Dequeue  2. Head는 Tail 추월  3. Tail Node 반환. |
|  | 1. 다른 스레드의 Enqueue  2. 메모리풀에서 TailNode 할당 받음.  3. AllocNode 초기화 후 TailNextNode로 갱신.  4. TailNextNode가 Tail이 되는 상황.  5. Capacity 상승. |

1. **버그리포트 2 : TailNextNode가 NULL이 아닌 상황 (1)**
   1. **로그 디버깅**

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **테이블이(가) 표시된 사진  자동 생성된 설명** |  | **테이블이(가) 표시된 사진  자동 생성된 설명** |
| 1. Tail Key : 107  2. 컨텍스트 스위칭 | 1. 다른 스레드에서 Tail 갱신  2. Key 갱신 (107 -> 108) | 1. 14044 스레드로 돌아와서 현재 시점의 TailNode Snapshot 생성.  2. TailNextNode NULL이라 판단. |
| **4** | **5** | **6** |
| **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** |
| 1. TailNextNode를 AllocNode로 갱신 | 1. 이전 Key와 현재 TailKey가 다름.  2. Tail 갱신 실패. | 1. Capacity 상승  2. 하지만 Tail 갱신은 실패.  3. TailNextNode가 NULL이 아닌 상황 발생. |

* 1. **시나리오**

|  |  |
| --- | --- |
|  | 1. SnapshotTail Key 세팅  2. 컨텍스트 스위칭 |
|  | 1. Thread B에서 Enqueue 완료  2. Tail Key 갱신 |
|  | 1. SnapshotTail Node 세팅  2. SnapshotTailNextNode를 NULL로 판단  3. SnapshotTailNextNode를 AllocNode로 갱신  4. SnapshotTail Key와 현재 TailKey 미일치.  5. Tail 갱신 실패. |

1. **버그리포트 3 : TailNextNode가 NULL이 아닌 상황 (2)**
   1. **로그 디버깅**

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** |
| 1. SnapshotTail 생성  2. 컨텍스트 스위칭 | 1. 22832스레드의 SnapshotTail이 다른 스레드의 EnQ, DeQ를 거치며 메모리풀에 반환.  2. 이후 13828 스레드의 Enqueue 작업에서AllocNode가 된 상황. | 1. AllocNoded 초기화  2. 컨텍스트 스위칭 |

|  |  |  |
| --- | --- | --- |
| **4** | - | - |
| **테이블이(가) 표시된 사진  자동 생성된 설명** |  |  |
| 1. 22832 스레드에서 SnapshotTailNext가 NULL이라 판단.  2. TailNextNode를 AllocNode로 갱신.  3. SnapshotTail이 현재 Tail과는 다른 상황.  4. Tail 갱신 실패.  5. Capacity 증가. | 6. 이후 13828 스레드로 돌아왔을 때 해당 스레드의 AllocNodeNext가 NULL이 아닌 상황  7. Tail을 AllocNode로 갱신.  8. TailNextNode가 NULL이 아닌 상황 발생. |  |

* 1. **시나리오**

|  |  |
| --- | --- |
|  | 1. Thread A SnapshotTail 생성  2. 컨텍스트 스위칭 |
|  | 1. Thread A의 SnapshotTail이 다른 스레드의 EnQ, DeQ를 거치며 메모리풀에 반환.  2. 이후 Thread B의 Enqueue 작업에서 AllocNode로 할당.  3. AllocNode 초기화  4. 컨텍스트 스위칭 |
|  | 1. ThreadA로 돌아왔을 때 SnapshotTail의 Next가 NULL이라 판단  2. SnapshotTailNextNode를 AllocNode로 갱신  3. 현재 Tail과 SnapshotTail이 일치하지 않음.  4. Tail 갱신 실패. |
|  | 1. Thread B로 돌아왔을 때 AllocNode의 Next가 Thread A로 NULL이 아닌 상황.  2. TailNextNode를 AllocNode로 갱신.  3. TailNextNode가 NULL이 아닌 상황. |

1. **버그 1,2,3 상황 해결**
   1. 버그리포트 1, 2, 3의 상황은 모두 TailNextNode가 NULL이 아닌 상황에서 발생했다.
   2. 하지만 Lock-Free Queue에서 버그리포트 2, 3의 상황은 에러로 보지 않는다. 즉, Dequeue 상황에서 버그리포트 2, 3의 상황인 경우해당 스레드가 Tail을 한 칸 밀고 Dequeue를 진행한다.

|  |  |
| --- | --- |
|  | 1. TailNextNode가 NULL이 아닌지 체크. |
|  | 2. 현재 Tail을 TailNextNode로 갱신. |
|  | 3. HeadNextNode를 Head로 갱신.  4. Capacity 차감. |

1. **버그리포트4. Dequeue에서 동일 값을 반환하는 상황**
   1. **로그 디버깅**

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** | **테이블이(가) 표시된 사진  자동 생성된 설명** |
| 1. HeadNextNode를 Head로 갱신.  2. 이전 Head 반환.  3. 컨텍스트 스위칭 | 1. 28948 스레드의 HeadNextNode 반환되는 상황. | 1. 다른 스레드의 Enqueue에서 HeadNextNode 를 할당받음.  2. HeadNextNode에 해당 스레드의 EnqData 세팅. |

|  |  |  |
| --- | --- | --- |
| **4** | **-** | **-** |
| **테이블이(가) 표시된 사진  자동 생성된 설명** |  |  |
| 1. 다시 28948 스레드로 돌아와서 HeadNextNode의 데이터 참조.  2. 하지만 이전에 획득했던 HeadNextNode 의 데이터와 다른 상황. |  |  |

* 1. **시나리오**

|  |  |
| --- | --- |
| **텍스트, 시계, 스크린샷이(가) 표시된 사진  자동 생성된 설명** | 1. HeadNextNode를 Head로 갱신.  2. 이전 Head 반환.  3. 컨텍스트 스위칭 |
|  | 1. 다른 스레드의 Enqueue에서 HeadNextNode 를 할당받음.  2. HeadNextNode에 해당 스레드의 EnqData 세팅. |
|  | 1. 다시 28948 스레드로 돌아와서 HeadNextNode의 데이터 참조.  2. 하지만 이전에 획득했던 HeadNextNode 의 데이터와 다른 상황. |

* 1. **해결**
     + 해당 에러는 Head를 먼저 갱신함으로 인해 다른 스레드가 Snapshot\_HeadNextNode에 접근할 수 있는 상황에서 발생했다.
     + 때문에 Head를 갱신하기 전에 데이터를 먼저 뽑고 Head를 갱신한다.

|  |  |
| --- | --- |
| **텍스트, 시계, 스크린샷이(가) 표시된 사진  자동 생성된 설명** | 1. HeadNextNode Data를 Destination에 세팅. |
| **텍스트, 시계, 스크린샷이(가) 표시된 사진  자동 생성된 설명** | 2. Head를 HeadNextNode로 갱신 |