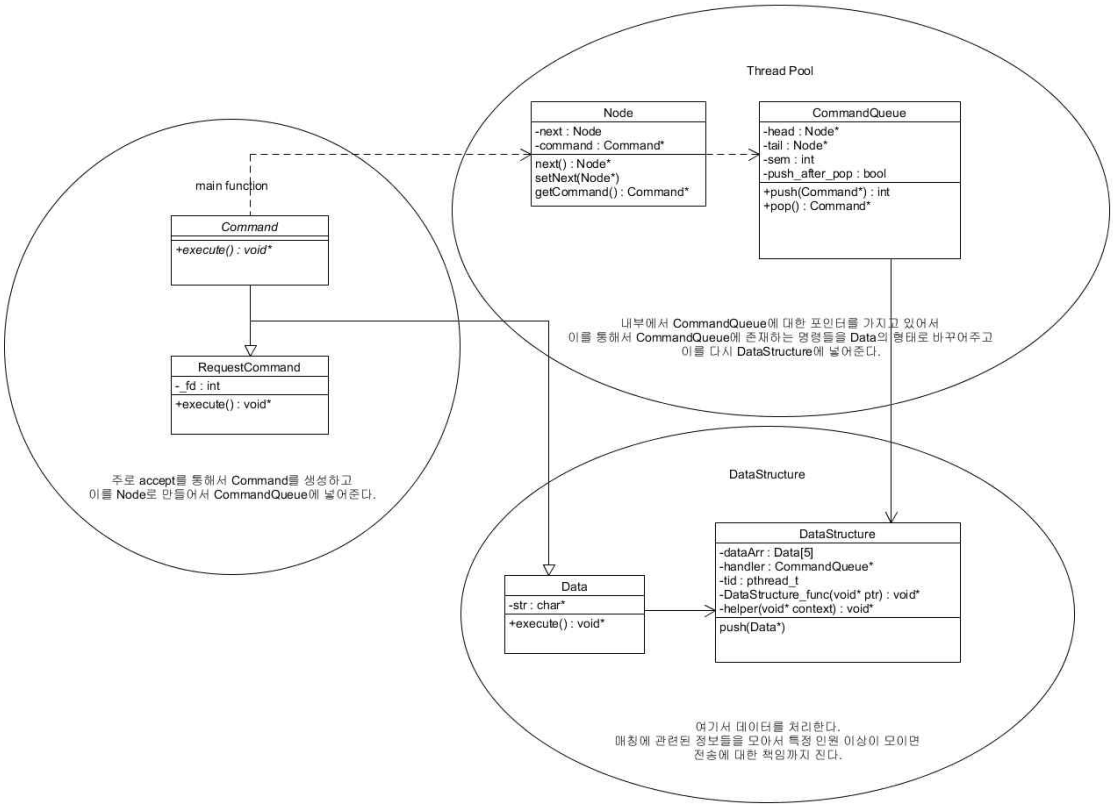
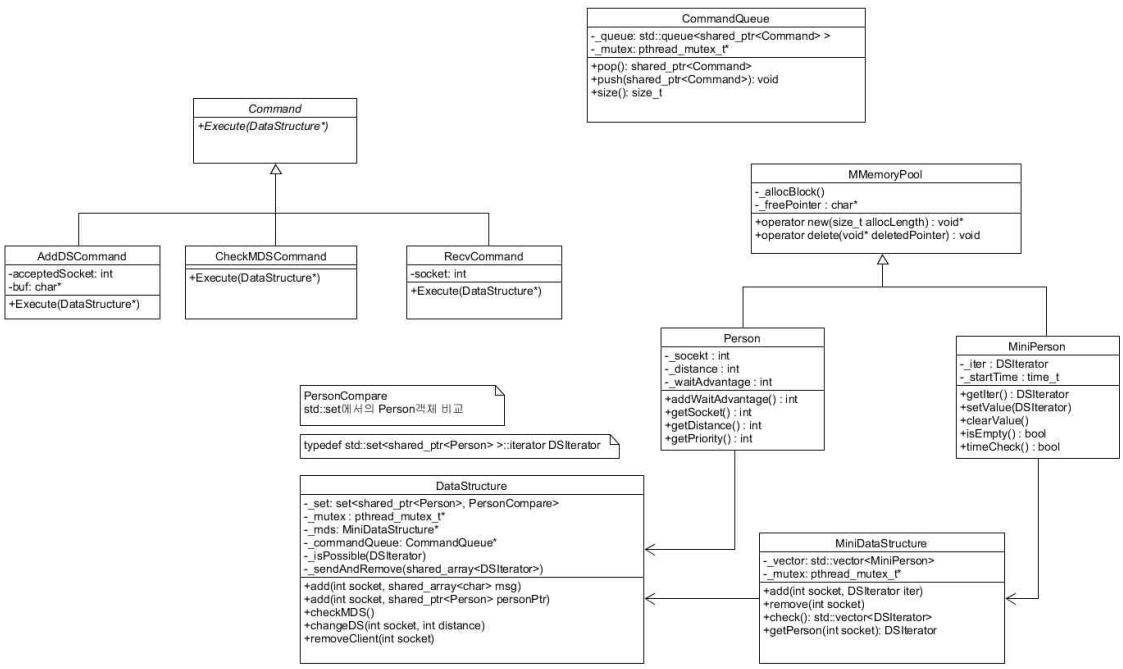
프로젝트명	Plan'T Server		
개발 인원	1명	개발 기간	2016.12 ~ 2017.03
개발 언어	C++		
개발툴	Vim		
작품 소개(요약)	Plan'T에서 기존에 사용하던 HTTP 서버를 대체하는 새로운 서버를 개발하고자 시작하였다. 환경은 Ubuntu 16.04에서 Vim을 통해 개발하였으며, POSIX에 따라 구현하였다.		
초기 개발 내용	(참고 자료1) Server를 C++로 구현하기 위해서 몇 번의 시행착오를 거쳤다. 처음에는 중요한 역할을 하는 모듈들에 대한 설계만 그리고 시작하였다. 그림 속 동그라미들이 모듈의 역할을 한다. Main Function에서는 accept만을 수행하고 이를 통해서 Node라는 CommandQueue의 Element를 만들고 이를 CommandQueue에 넣어준다. Thread Pool은 이미 만들어져있는 CommandQueue를 공유하며 접근한다. Thread Pool에서는 CommandQueue에 존재하는 Command를 실행하여준다. 주로 하는 역할은 socket에서 읽어들이고 이러한 정보를 DataStructure에 넣어준다. DataStructure에 데이터를 넣을 때에는 DataStructure내부에 있는 CommandQueue를 통해서 넣어준다. DataStructure에서는 자료구조를 이루고 있으며 이것만을 처리하는 Thread가 존재하여 class 내부에 있는 CommandQueue에서 명령을 받아서 처리한다. 그리고 Matching이 가능해지면 이에 대한 정보를 Client에 전송하는 책임까지 지니고 있다. 구현시의 목표는 모듈별 분리를 통해서 각각 독립적으로 작동하도록 하는 것이었다. 이를 통해서 사용자의 편의 맞게 DataStructure를 변경할 수 있고, Command를 추가할 수 있도록 하고자 하였다. 또한, Read연산을 Thread Pool에서 하도록 함으로써 네트워크 연결의 문제등으로 Read가 지연되는 경우에도 Latency없이 작동하도록 설계하였다.		
초기 개발 문제점	하지만 초기 개발에는 문제점이 많았다. 우선, 모듈로 이루어져서 독립적으로 작동은 하지만 이에 따른 오버헤드가 존재했다. Thread Pool에서 CommandQueue에 동시접근을 막기위해서 Mutex를 사용하였다. 또한, DataStructure내부에서도 CommandQueue가 존재해서 Thread Pool에서 이 queue에 정보를 넣을 때와 DataStructure에서 처리를 위해 pop할 때에도 이를 Mutex가 사용된다. 즉, 모듈마다 전부 Mutex를 통해서 보호되어져야 한다. 또한, 실제로 처리되는 데이터보다 이러한 Mutex 처리에 들어가는 비용이 더 크다고 판단하였다. 또한, 설계를 진행 할 때, 모듈만으로만 클래스 다이어그램을 구성하는 바람에 구체적인 구현을 할 때에 바뀌는 사항들이 굉장히 많았다.		
개발 내용	초기 개발의 문제점들을 보완하기 위해서 새롭게 서버를 구현하였다. 우선 초기 개발의 큰 문제점 중 하나인 Mutex에 대한 오버헤드에 관해서는 멀티쓰레드 환경으로 구현하여야 하기 때문에 꼭 필요하기는 하지만 독립적인 모듈로 구성하면서 Mutex의 사용이 더 늘어났다고 판단하여 이번에는 독립적인 모듈이 아닌 서버를 하나의 독립적인 형태로 구성하였다.		

	또한, 시간이 지남에 따라서 추가되는 가중치를 효과적으로 계산하기 위해서 새로운 Command도 고려대상에 넣었으며, 클라이언트의 새로운 요청이 왔을 때의 경우도 생각하였다. Command라는 abstract Class가 존재하고 이를 상속받는 3개의 Command로 구성하였다. 이러한 형식은 기존에 있던 방식으로 Command를 필요에 따라서 쉽게 확장 할 수 있기 때문에 그대로 사용하였다. CommandQueue 역시 동일한 방식으로 동작한다. 하지만 기존에는 Queue를 직접 구현하였다면 안정성 측면을 고려하여 std::queue를 사용하였다. 그리고 불필요한 Node class를 없애고 대신 Command를 직접 queue에 넣는 방식으로 구현하였다. DataStructure도 시간이 지남에 따라서 가중치를 쉽게 계산하기 위해서 따로 MiniDataStructure를 두었다. 이러한 방식을 사용해서 모든 Person객체에 대한 탐색이 DataStructure에서의 Mutex 자원을 오래 차지하고 있는 것을 줄여줄 것으로 예상된다. MMemoryPool Class는 이름 그대로 Memory Pool을 지원해준다. Command의 경우에는 각 Command의 데이터 크기가 모두 동일하지 않기 때문에 Memory Pool을 이용하는데 어려움이 따르지만 Person이나 MiniPerson Class의 경우에는 자주 생성되고 소멸되고 그 크기가 일정하기 때문에 Memory Pool을 이용하기에 적당하다. 하지만 operator new와 operator delete로 구성되어져 있기 때문에 멀티 쓰레드 환경에서 주의해야 한다. 하지만 Person의 경우는 DataStructure의 Mutex에 의해서 보호를 받고 MiniPerson의 경우에는 MiniDataStrcture의 Mutex에 의해서 보호되기 때문에 동시에 호출될 일은 없다. 그리고 대부분의 경우의 boost::shared_ptr이나 boost::shared_array과 같은 자원 관리 클래스를 이용하여 메모리 누수를 예방하였다.
문제점/해결방안	우선, 데이터베이스에 대한 고려가 없었다. 이용자를 모두 Person과 MiniPerson을 이용하여 메모리에 저장하기 때문에 메모리가 적은 환경이나 이용자가 많이 들어오게 되면 메모리가 부족하게 된다. 이를 처리하기 위해서는 데이터베이스를 구성하여야 한다. 데이터베이스를 구성한다면 기존의 DataStructure는 캐시로써 사용하고 이후에 DataStructure에서 데이터베이스의 데이트를 추가적으로 불러오거나 쓰는 함수가 필요할 것으로 보인다. 또한, 데이터베이스가 추가된다면 MiniDataStructure의 필요성도 줄어들 것으로 보인다. 혹은 또 다른 프로그램을 작성하여 DataStructure Class를 데이터베이스에 기반하여 새로 작성하는 것도 간편하게 구현가능한 방법이다. 하지만 사용자 한 명당 Person 객체 하나와 MiniPerson 객체를 하나씩만 사용하므로 메모리는 총 20Byte정도 밖에 들지 않기 때문에, 적은 사용자가 예상된다면 응답시간이 빠른 이 방식을 사용하는게 옳고 그렇지 않다면 데이터베이스를 구성하는 것이 옳다.
Github 주소	https://github.com/kgp1202/matching-original.git https://github.com/kgp1202/matching3.git


Thread Pool
Node
-next : Node
-command : Command*
next() : Node*
setNext(Node*)
getCommand() : Command*
CommandQueue
-head : Node*
-tail : Node*
-sem : int
-push_after_pop : bool
+push(Command*) : int
+pop() : Command*
main function
Command
+execute() : void*
RequestCommand
-_fd : int
+execute() : void*
내부에서 CommandQueue에 대한 포인터를 가지고 있어서
이를 통해서 CommandQueue에 존재하는 명령들을 Data의 형태로 바꾸어주고
이를 다시 DataStructure에 넣어준다.
주로 accept를 통해서 Command를 생성하고
이를 Node로 만들어서 CommandQueue에 넣어준다.
DataStructure
DataStructure
-dataArr : Data[5]
-handler : CommandQueue*
-tid : pthread_t
-DataStructure_func(void* ptr) : void*
-helper(void* context) : void*
push(Data*)
Data
-str : char*
+execute() : void*
여기서 데이터를 처리한다.
매칭에 관련된 정보들을 모아서 특정 인원 이상이 모이면
전송에 대한 책임까지 진다.

(참고 자료1)


CommandQueue
-_queue: std::queue<shared_ptr<Command> >
-_mutex: pthread_mutex_t*
+pop(): shared_ptr<Command>
+push(shared_ptr<Command>): void
+size(): size_t
Command
+Execute(DataStructure*)
MMemoryPool
-_allocBlock()
-_freePointer : char*
+operator new(size_t allocLength) : void*
+operator delete(void* deletedPointer) : void
AddDSCommand
-acceptedSocket: int
-buf: char*
+Execute(DataStructure*)
CheckMDSCommand
+Execute(DataStructure*)
RecvCommand
-socket: int
+Execute(DataStructure*)
Person
-_socekt : int
-_distance : int
-_waitAdvantage : int
+addWaitAdvantage() : int
+getSocket() : int
+getDistance() : int
+getPriority() : int
MiniPerson
-_iter : DSIterator
-_startTime : time_t
+getIter() : DSIterator
+setValue(DSIterator)
+clearValue()
+isEmpty() : bool
+timeCheck() : bool
PersonCompare
std::set에서의 Person객체 비교
typedef std::set<shared_ptr<Person> >::iterator DSIterator
DataStructure
-_set: set<shared_ptr<Person>, PersonCompare>
-_mutex : pthread_mutex_t*
-_mds: MiniDataStructure*
-_commandQueue: CommandQueue*
-_isPossible(DSIterator)
-_sendAndRemove(shared_array<DSIterator>)
+add(int socket, shared_array<char> msg)
+add(int socket, shared_ptr<Person> personPtr)
+checkMDS()
+changeDS(int socket, int distance)
+removeClient(int socket)
MiniDataStructure
-_vector: std::vector<MiniPerson>
-_mutex: pthread_mutex_t*
+add(int socket, DSIterator iter)
+remove(int socket)
+check(): std::vector<DSIterator>
+getPerson(int socket): DSIterator

(참고 자료2)