**Packet Generator**

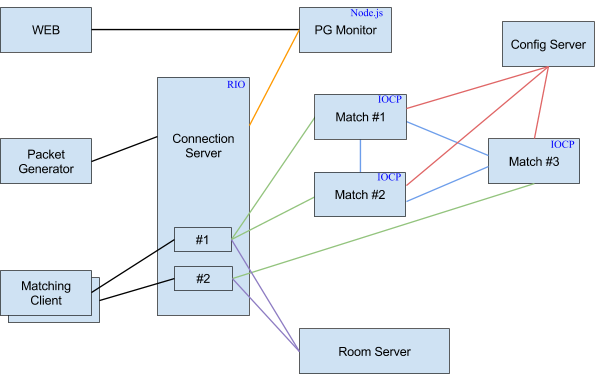
**& Matching Project**

Version 1.0.0.0

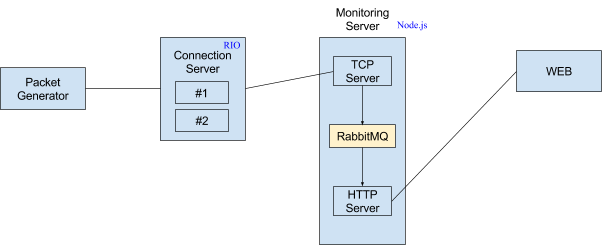
[버전 관리]

|  |  |  |  |
| --- | --- | --- | --- |
| Version | 날짜 | 변경 내역 | 이름 |
| 1.0.0.0 | 2016.11.09 | 초안 | 심혜진 |
| 1.0.0.1 | 2016.11.09 | Protocol에 Command 추가 | 이동규 |
| 1.0.0.2 | 2016.11.10 | Protocol Command 표 수정 | 심혜진 |
| 1.0.0.3 | 2016.11.22 | Protocol body, Command 수정 | 심혜진 |
|  |  |  |  |
|  |  |  |  |

구조도



Packet Generator



1. Packet Generator는 사용자로부터 보낼 패킷의 수, 패킷의 길이, 프로토콜(TCP/UDP)를 입력 or Config로 받는다.
2. Connection Server로 TCP 연결을 한다.
3. PG Monitor에게  보 낼 총 패킷 수와 시작을 알리는 메세지(PG\_START)를 보낸다.
4. PG Monitor로 응답을 받으면 사용자 정의에 따라 TCP/UDP Socket으로 Packet을 보낸다.
5. PG Monitor에게  끝을 알리는 메세지(PG\_END)를 보낸다.
6. 최종 출력 값  
   - 패킷을 보내는데 걸린 시간  
   - 총 보낸 패킷 수  
   - 초당 보낸 패킷 수

PG Monitor

**# 구성**

1. PGM Server는 Node.js로 개발한다.
   1. TCP : Net module
   2. HTTP : Express module
   3. log : log4net
   4. data share : RabbitMQ
2. PGM Server는 2개의 Connection을 가진다.
   1. TCP : Connection Server
   2. HTTP : Web Client
3. 그에 따라 PGM Server는 TCP Server, HTTP Server로 구성된다.
4. Monitor라는 객체가 Performance 지표들을 관리하고, 이를 통해 TCP Server와 HTTP Server 간에 데이터를 공유한다.
5. Web Client가 x초 마다 (default: 5s) performance 지표들을 Web Server에 요청한다.
6. Packet Generate Client가 종료되면, 전체적인 log들을 Web Client에 파일로 제공한다.

**# Flow**

<TCP Server>

1. TCP Server는 Packet Generate Client로부터 PG\_START 패킷(+전송할 패킷 수)을 받는다.
2. TCP Server는 PG\_START message를 Publish하여 모니터링 시작을 알린다. (to HTTP Server)
3. TCP Server는 packet 수를 통해 PPS를 계산한다.
4. TCP Server는 1초마다 새롭게 계산한 PPS를 HTTP Server에 Publish한다.
5. TCP Server는 Packet Generate Client로부터 받은 PG\_ENd 패킷을 Publish하여 HTTP Server에 모니터링 종료를 알린다.

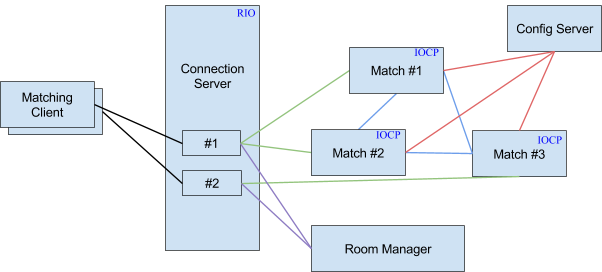
<HTTP Server>

1. HTTP Server는 PG\_START message를 subscribe하여 모니터링을 시작한다.
2. HTTP Server는 TCP Server가 보내는 PPS를 Subscribe한다.
3. PPS데이터들은 파일 로그로 기록한다.
4. HTTP Server는 Client가 특정주기마다(default: 5s)  요청하는 from ~ to 시간 동안의 PPS를 제공한다.  
   ex) from: 5, to : 8  
   result : [  
    {“duration” : 5 , “packets” : 50000}, // 5s ~ 6s 사이에 50000 packet이 전송됨.  
    {“duration” : 6 , “packets” : 20000}, // 6s ~ 6s 사이에 20000 packet이 전송됨.  
    {“duration” : 7 , “packets” : 50000},  
    {“duration” : 8 , “packets” : 50000}  
   ]
5. 테스트가 끝나면 해당 로그 기록을 파일로 저장한다.

WEB

1. x초의 간격으로 UI를 update한다. (default : 5s)
2. PPS 데이터들은 d3.js 차트로 시각화한다.
3. 테스트 관련 로그 파일을 다운로드 할 수 있다.

Matching



Matching Data Flow

1. connection server에 room server와 matching server가 붙는다.
2. connection server는 client를 받는다.
3. client가 매칭 요청을 matching server로 보낸다.(CS를 통해)
4. matching server들은 client의 stat을 보고 비슷한 client와 매칭을 한다.
5. matching server가 room server에게 매칭 결과를 보낸다.
6. room server는 매칭 결과로 방을 만들고 matching server에 결과를 보낸다.
7. matching server는 client에게 매칭완료를 알린다.
8. client는 room server에게 방 참가를 요청.
9. room server는 client에게 game start 전송.
10. 방 참가 10~20 초 후 room server는 room 내의 client들에게 game end 메시지 전송.
11. 3으로 돌아감.

Matching Client

1. 목적이 우리 Network Engine을 테스트 위해서 자동적으로 Client Emulation 한다.
2. 나중에 Game Client들이 Unity를 사용 하기 위해서:
   1. **C#**하고 **.NET 3.5**으로 개발한다.
   2. **SuperSocket.ClientEngine**으로 개발한다.
3. Matching Client가 Connection Server (지금 CS#1이나 CS#2)와 TCP로 연결 한다.
4. 연결하기 후에 Game Emulation 한다:
   1. Loop: Matching 요청; Response 받았을 때 Success이면 룸 조인 요청; 룸 조인 응답이 오면 waiting; Game Start가 오면 waiting; Game End가 오면 응답을 함.
   2. Disconnect 하면 다시 연결 한다. 재 연결 할 때 Game State을 저장 의해서 Game 계속 할 수 있다.
5. Config file로 CS의 IP/Port하고 Game Emulation의 Instructions/Variables를 받는다.

Connection Server

목적이 Game Client나 Server한테 연결해서 Client나 Server의 메세지를 Relay 한다.

1. High-throughput Network Performance 위해서:
   1. C++로 개발한다.
   2. Windows의 RIO로 개발한다.
2. Config file로 RIO/IOCP configuration 값을 받는다.
3. Classes
   1. RIOManager - RIO/IOCP/WinSock을 다 Manage 한다.
      * BufferManger에게 RIO-Extention-Functions을 줘서 BufferManager로 Buffer를 받는다.
      * 3가지 Socket (UDP Socket, TCP Listener, TCP Socket)을 만든다.
      * IOCP Queue와 RIO\_CQ를 만들어서 Socket 기능과 연결 한다. UDP Socket과 TCP Socket은 RIO\_RQ 가지고 RIO\_CQ하고 연결 한다.
      * TCP Listener은 AcceptEx로 얻은 socket을 IOCP Queue하고 연결 한다. RIO\_CQ는 IOCP Queue하고 연결 한다.
      * Failed I/O 되면 그 Socket을 close한다.
   2. BufferManager - VirtualAllocEx로 큰 메모리를 할당하고, RIO Buffer Registration을 하여 RIO buffer를 만든다. 이 메모리를 메시지들의 최대 크기를 고려해 일정한 크기로 잘라서 관리하며, rio manager에게 새로운 버퍼를 얻거나 쓴 버퍼를 free 하는 API를 제공한다.
   3. ProcessManager - RIOManager에게서 completion results (EXTENDED\_RIO\_BUF)을 받아서 처리한다.
      * 메세지의 header를 읽어서 메세지의 destination을 계산 한다. 나중에 header을 modification 기능하기도 한다.
      * Instructions (Send, Receive, Free Buffer, Close Socket) vector를 만들어서 RIOManager에 준다.
4. Application Frame
   1. Stage 1: Application Setup - RIOBuffers/IOCP Queue, RIO\_CQ, UDP Socket, TCP Listener를 시작 된다.
   2. Stage 2: Initiation - 비동기 RIOReceive와 AcceptEx를 posting 한다. Thread Pool을 만들어서 Thread Function 시작 된다.
   3. Stage 3: Thread Function:
      * Proactor - GQCS으로 Completion 나올 때까지 기다려서 Completion Key를 받는다. Completion Key로 RIO completion, accept function, quit function, information request function을 demultiplexing 한다.
      * Completion Handling - RIOManager가 AcceptEx Completion을 handle 한다. RIO completion 나면 RIOManger가 RIO\_CQ에게서 RIO\_RESULT을 받는다.
      * RIOManager는 RIO\_RESULT의 Request Context으로 EXTENDED\_RIO\_BUF를 얻어 이것을 반환한다.
      * ProcessManger는 이 buf들로 작업한 결과를 반환한다.
      * Initiator - Completion Handling 후에 ProcessManager가  RIOManager에게 instruction을 줘서 RIOManager가 새로운 비동기 Operation (AcceptEx, RIOReceive, RIOSend) 시작 된다.
   4. Stage 4: Main thread가 user command을 받을 때까지 기다리고 있다. 일단 Quit Command을 있다. Quit Command하면, IOCP에 Quit Message을 Thread에게 보낸다. Thread들이 다 끝나서 RIOManager로 Resource Clean-up 한다.
5. Source/Destination Management:
   1. Client가 Connection Server에게 메세지 보낼 때, Header에서 Destination 값이 있다. Server에게 Relay 하기 전에 ProcessManager가 Source 값을 RIO\_RQ의SocketContext으로 바꾼다. Destination 값이 불특정 (예: Matching\_Server\_1 말고, Matching\_Server\_Any) 해서, RIOManager가 Round-robin으로 Server에게 보낸다.
   2. Server가 Connection Server에게 메세지 보낼 때, Header에서 Source 값하고 Destination 값(= RIO\_RQ의SocketContext)이 있다. ProcessManager가 받는 buffer부터 그 Destination값을 읽어서 RIOManager에게 준다. RIOManager가 SocketContext (KEY)/RIO\_RQ (VALUE)의 hash 가지고 무슨 RIO\_RQ에 메세지를 Relay 하는 Destination 알아진다. 이렇게 RIOManager가 맞는 Destination에 메세지를 Relay 할 수 있다.
   3. RIOManager는 세개의 hash를 (Client, Matching Server, Room Manager) 가진다. Client, MS, RM은 서로 다른 포트로 accept 하며, 보안 문제가 있을 시 나중에 추가. Connection Server와 연결을 시작 할 때 Socket Handle 값으로 SocketContext의 int 만들어서 hash에 넣는다.

Matching Server

1. IOCP를 사용하여 비동기 socket io를 처리한다.
   1. 모든 socket은 iocp에 등록하고 관리한다.
2. Config 를 받아온 후 초기화 작업을 한다.
   1. Config Server에 연결한다.
   2. MS List Request 보낸다.
   3. MS List를 받는다.
   4. Config Server에게 받은 Matching Server List 토대로 Matching Server에 접속한다.
3. Connection Server에 연결한다.
4. Match Server
   1. AcceptEX를 사용하여 Matching Server를 받는다.
   2. Match Server는 In-Memory를 사용하여 Waiting Client Pool을 관리한다.
      * Client Pool 구성 요소
        1. Client ID
        2. Client metric  총 합
   3. Matching Server에 Matching Request가 도달하면 Matching을 시도한다.
   4. Latency
      * 특정 interval마다 MS 각각은 다른 MS들 간의 latency를 랜덤으로 설정한다.
      * latency는 0.001 ~ 1.000 사이의 값을 가진다.
   5. Metric
      * Matching Server는 자기에게 Matching Request 들어온 Client들의 정보와 broadcasting된 client들의 정보를 식별할 수 있다.
      * Matching Client Metri은 Offense, Defense, Level로 구성된다.
      * Matching Request가 들어오면, Offense, Defense, Level을 Random Generate.
      * 생성한 Offense, Defence, Level에 가중치를 주어 Metric을 계산한다.
      * Offense : 1 ~ 99 \* 가중치(미정)
      * Defence : 1 ~ 99 \* 가중치(미정)
      * Level : 1 ~ 99 \* 가중치(미정)
      * Metric을 생성 후, Waiting Client Pool에서 상대를 찾는다.
      * 자신의 Waiting Client Pool에 알맞은 상대가 없다면, Client의 Matching Request를 Metric과 함께 다른 Matching Server에 broadcasting한다.
      * 다른 서버에게 받은 클라이언트 정보로 매칭 상대를 정해서 응답한다.
      * 다른 서버에도 매칭 클라이언트가 없을 시 Waiting Client Pool에 넣는다.
   6. Matching
      * Matching은 다음과 같은 요소들을 고려한다.
        1. latency
        2. Client metric
        3. waiting time : 고려 중
      * latency : 0.2 이상이면 matching 불가
      * Client Metric에서 특정 범위 이내여야 matching. (추후 밸런스 조정)
      * Matching 할 Client pair를 구성하면, 이를 Room Server에 알려준다.(Client ID, Client ID)
      * Room Server의 응답에 따라 재 매칭을 시도하거나, 매칭 완료 결과를 Matching Client에게 전송한다.
      * matching이 완료되면 Matching Server의 Client Pool에서 제거한다.

Config Server

Matching Server의 Config를 관리하기 위해 사용한다.

1. Config Server는  Matching Server로부터 연결을 받는다.
2. Matching Server 리스트를 관리한다.
   1. Matching Server가 Connection 되면 리스트에 추가
   2. Matching Server와  Disconnection 되면 리스트에 해당 socket 삭제
   3. 주기적 Health Check로 Matching Server 관리
3. 연결된 MS를 비동기 Receive한다.
   1. Match Server에게 Request를 받으면 기존에 접속 되어 있는 Matching Server List를 보낸다.

Room Server

룸을 관리하고 게임을 시작, 종료한다.

**전체 흐름**

1. Matching Server로부터 매칭된 두 client의 정보를 받는다.
2. 클라이언트와 겹치는 곳이 없는지 확인하고 방을 생성한 뒤 결과를 MS에게 보낸다.
3. client의 요청을 받아 room join한다.
4. client에게 게임 시작을 알린다.
5. 10~20초 대기
6. client에게 게임 종료를 알린다.

상세

1. Receive를 동시에 많이 할수 있도록 하기 위해 IOCP 사용.
2. 시작과 동시에 스레드 개수 만큼 Receive를 호출한다.
3. IOCPManager가 모든 데이터 recv, send를 관장하며, 받은 데이터는 message queue에 저장하고 RoomServer는 이것을 꺼내 쓴다.

Protocol 정의

struct Body

{

COMMAND command;

STATUS status;

string data1;

string data2;

};

struct Header

{

int length;

SrcDstType srcType;

SrcDstType srcCode;

SrcDstType dstType;

SrcDstType dstCode;

}

|  |  |
| --- | --- |
| header | (int) length |
| (int) src type |
| (int) src code |
| (int) dst type |
| (int) dst code |
| (flatbuffers) body | |

enum TERMINALTYPE

{

MATCHING\_SERVER = 0,

MATCHING\_CLIENT,

ROOM\_SERVER,

PACKET\_GENERATOR,

MONITORING\_SERVER,

CONFIG\_SERVER,

CONNECTION\_SERVER

};

enum COMMAND

{

//Common

HEALTH\_CHECK = 0,

//MS~MS

NOTI\_MATCH\_REQUEST = 10,

NOTI\_MATCH\_SUCCESS,

LATENCY,

//MS~Client = 20,

MATCH\_REQUEST,

MATCH\_RESPONSE,

//MS~Config

MSLIST\_REQUEST = 30,

MSLIST\_RESPONSE,

//Room~MS

ROOM\_CREATE\_REQUEST = 40,

ROOM\_CREATE\_RESPONSE,

//Room~Client

ROOM\_JOIN\_REQUEST = 50,

ROOM\_JOIN\_RESPONSE,

GAME\_START,

GAME\_END

//PG~Monitor

PG\_START = 60,

PG\_END,

PG\_DUMMY

};

enum STATUS

{

SUCCESS = 0,

FAIL,

NONE

};

struct AddressInfo

{

SrcDstType srcType;

int srcCode;

SrcDstType dstType;

int dstCode;

};

Struct 유저정보

{

int ID;

int metric;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| no | cmd | data | status | 비고 |
| 0 | HEALTH\_CHECK\_REQUEST |  | NONE |  |
| 1 | HEALTH\_CHECK\_RESPONSE |  |  |  |
| 10 | NOTI\_MATCH\_REQUEST | 유저정보(client id, metric) | NONE | MS~MS |
| 11 | NOTI\_MATCH\_RESPONSE, | 유저정보(client id) | **NONE** | MS~MS |
| 20 | MATCH\_REQUEST |  | NONE | Client ~ MS |
| 21 | MATCH\_RESPONSE | 매칭정보 (client id, client id) | SUCCESS/FAIL | MS ~ Client |
| 30 | MSLIST\_REQUEST |  | NONE | Config ~ MS |
| 31 | MSLIST\_RESPONSE | matching server list (server id, ip) | SUCCESS FAIL | Config ~ MS |
| 32 | MS\_ID |  |  |  |
| 40 | ROOM\_CREATE\_REQUEST | (client id, client id) | NONE | MS ~ RM |
| 41 | ROOM\_CREATE\_RESPONSE | room num (on success) | SUCCESS FAIL | MS ~ RM |
| 50 | ROOM\_JOIN\_REQUEST | room num | NONE | MC ~ RM |
| 51 | ROOM\_JOIN\_RESPONSE |  | SUCCESS FAIL | MC ~ RM |
| 52 | GAME\_START |  | NONE | MC ~ RM |
| 53 | GAME\_END |  | NONE | MC ~ RM |
| 60 | PG\_START |  | NONE | PG ~Monitor |
| 61 | PG\_END |  | NONE | PG ~Monitor |
| 62 | PG\_DUMMY |  | NONE | PG ~Monitor |