경희대학교 전자공학과

2020년 2학기 졸업 논문

제목 :

유니티를 이용한

포톤 클라우드에서의 RTT측정

송은빈

**<유니티 선택 배경>**

여러 게임 개발 프로그램 중 사용자 층이 다양한 유니티는 직관적인 인터페이스들과 풍부한 정보가 특징이며 따라서 게임 초보자들도 이를 이용해 게임 개발이 용이하다. 또한 자체 에셋 스토어에서 제공하는 각종 리소스, 스크립트, 플러그인을 통해 초보자에게 어려울 수 있는 디자인, 모델링의 영역에 대한 부담감을 덜 수 있다.

**<네트워크 선택 배경>**

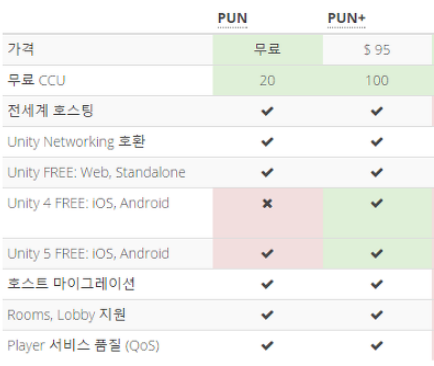
여러 네트워크 서버 중 구글 클라우드 서버, 아마존 웹 서비스, 마이크로소프트 Azure 중 가장 이 프로젝트와 알맞은 네트워크를 조사했다. 설계 단계에서 설계 비용에 제한이 있으므로 무료 오픈 소스를 중점적으로 조사했다. 여러 네트워크 중 Photon PUN2를 이용해 네트워크를 개발하기로 했다.

PUN은 Photon Unity Networking의 약칭이다. Unity 상의 멀티플레이어 네트워크 리얼타임 온라인기능을 강화한 키트로서 기존 Unity Networking과도 호환되며, Unity의 Asset Store에서 PUN 2 - FREE를 다운받아 import해서 쉽게 사용할 수 있다. PUN은 로우레벨부터 완전히 새로 제작되었으며 Photon Cloud/Server를 게임 백엔드로 사용했기 때문에 개발하기에 상당히 편리하고, 유연한 매치메이킹을 통해 플레이어들은 객체들이 네트워크를 통해 동기화될 수 있는 룸으로 이끌어 준다. 빠르고 (선택적으로) 신뢰할 수 있는 커뮤니케이션은 전용 Photon 서버를 통해 이루어지므로 클라이언트들은 1 대 1 연결이 필요하지 않다.

PUN은 기본적으로 Unity가 지원하는 모든 플랫폼으로 엑스포트되며 다음 두 가지 옵션이 있다.



PUN과 PUN+의 차이는 다음과 같다.



PUN의 경우 동시접속자 수가 20명까지 접속할 수 있으며, IOS와 Android를 제외하고 실행 가능하고, PUN+는 동시접속자 수가 100명까지 가능하며, 실행에 제한이 없다. 우리 게임의 경우에는 많은 동시접속자가 필요하지 않고, 컴퓨터를 이용한 게임이기 때문에 PUN을 사용했다.

포톤 사이트에서 계정을 생성하게 되면 Photon Cloud 애플리케이션을 위해 발급된 식별자인 AppID를 얻을 수 있는데, 게임에 AppID를 넣음으로써 게임에서 포톤 클라우드에 접속할 수 있게 해준다.

**<매트랩 선택 배경>**

매트랩은 수치 계산 환경의 [그래픽](http://www.ktword.co.kr/abbr_view.php?nav=&m_temp1=4975&id=46) 시각화 및 [프로그래밍](http://www.ktword.co.kr/abbr_view.php?nav=&m_temp1=5710&id=500)이 가능한 [소프트웨어](http://www.ktword.co.kr/abbr_view.php?nav=&m_temp1=3089&id=518)로 벡터, 행렬을 기본으로 하며 수치해석, 행렬연산, 신호처리, 모델링, 시뮬레이션 등을 통합, 구현시켜 계산환경을 시각적으로 제공해준다. 여러 다양한 계산 환경에서 이용할 수 있으며, 대화형 수치계산 및 그래픽 시각화를 가능하게 한다. 또한, 비교적 용이하게 새로운 명령어 및 함수를 만들어 확장할 수 있으며 프로그래밍이 가능하다는 특징이 있다.

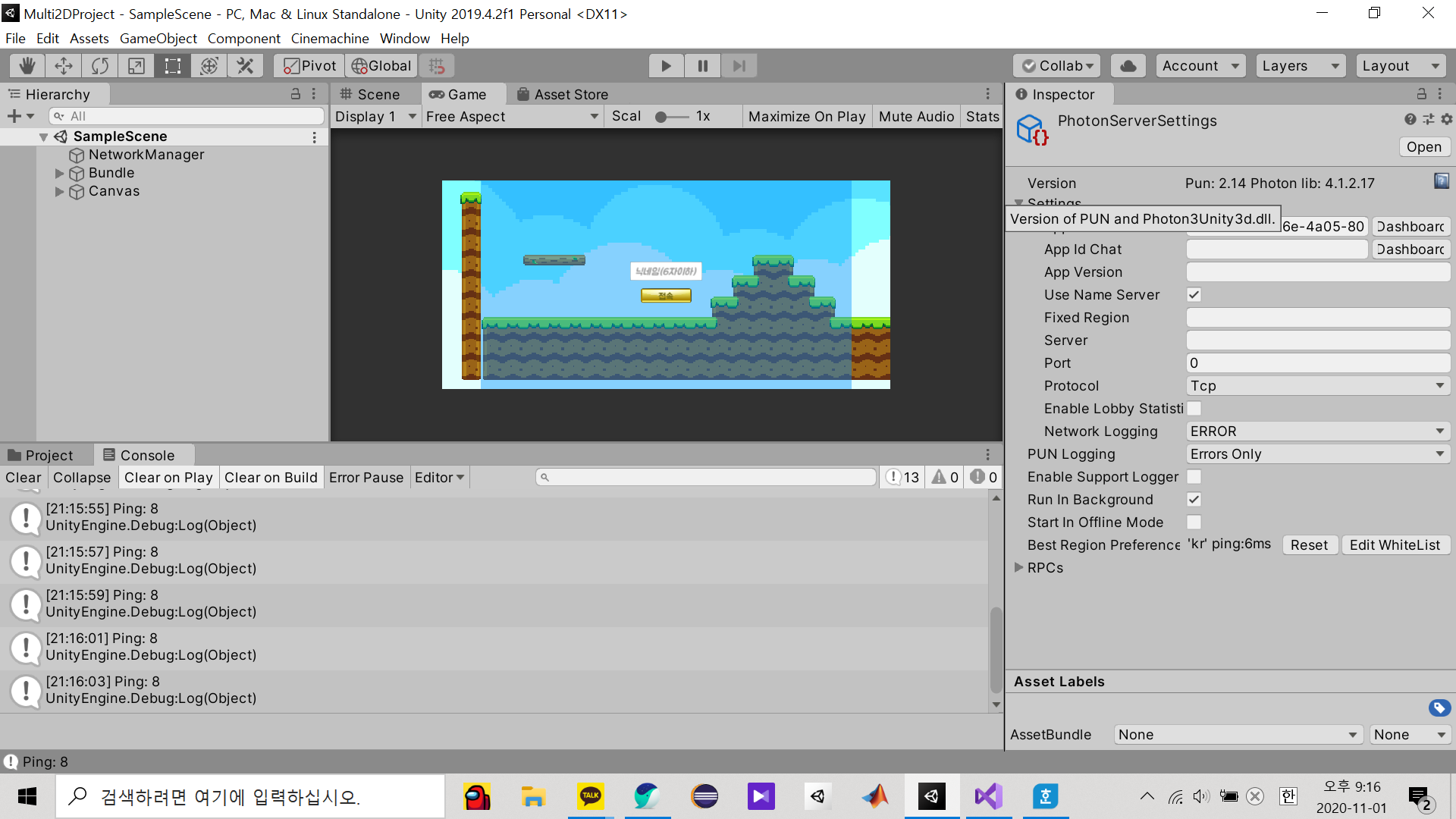
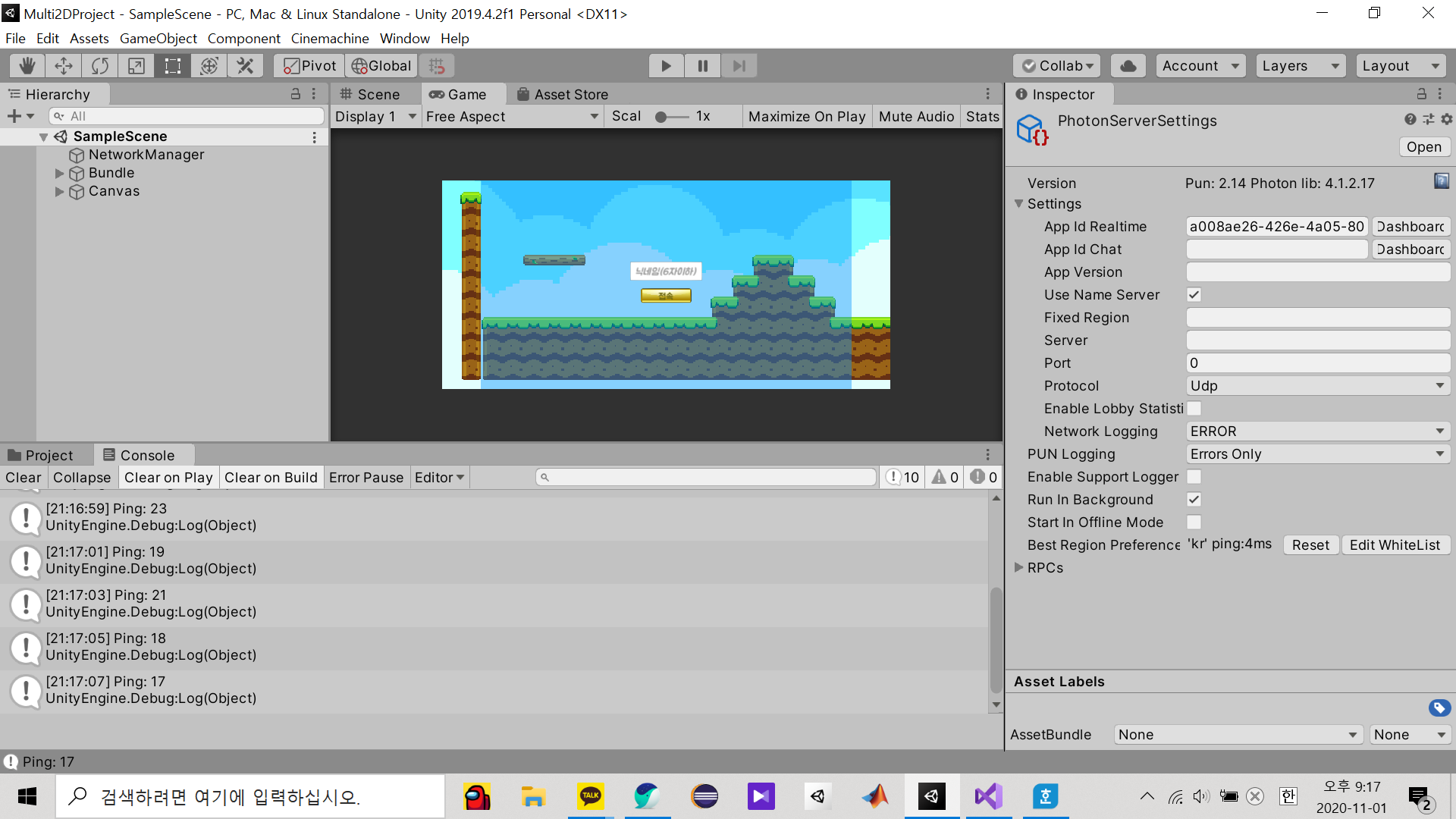
**<설계>**

핑(Ping)이란 Packet Internet Groper의 약어이며, 컴퓨터 네트워크 상태를 점검, 진단하는 명령이다. ping의 기본적인 작동 원리는 네트워크 상태를 확인하려는 대상(target)컴퓨터를 향해 일정 크기의 패킷(packet, 네트워크 최소 전송단위)을 가지는 메세지(ICMP echo request)를 보낸 후 이에 대해 응답하는 메시지(ICMP echo reply)를 보내면 이를 수신, 분석하여 대상 컴퓨터까지 도달하는 네트워크 상태가 어떠한 지 파악할 수 있다.

게임에서 RTT(Round Trip Time)를 측정하기 위해서 ping을 보내는 방법을 선택했다. ping을 측정할 때는 PUN에서 제공하는 PhotonNetwork.GetPing()을 활용하여 실시간 패킷 왕복 시간을 반환할 수 있는데, 이는 클라이언트에서 서버로 전송한 후 다시 클라이언트로 전달한 수치로 해석할 수 있다.

포톤 클라우드를 이용해서 보낸 ping의 값은 다음과 같다.

- TCP - UDP

임시로 특정 지역에 접속하기 위해서 PhotonNetwork.ConnectToRegion(region)으로 선택한 [Photon](https://doc-api.photonengine.com/ko-kr/pun/current/namespace_photon.html)클라우드 지역으로 접속할 수 있다.

PhotonNetwork.ConnectToRegion(region)을 사용하기 위한 접근 방법이 두 가지이다.

1. PhotonNetwork.NetworkingClient.AppId=PhotonNetwork.PhotonServerSettings.AppSettings.AppIdRealtime;

PhotonNetwork.ConnectToRegion(region);

1. PhotonNetwork.ConnectUsingSettings();

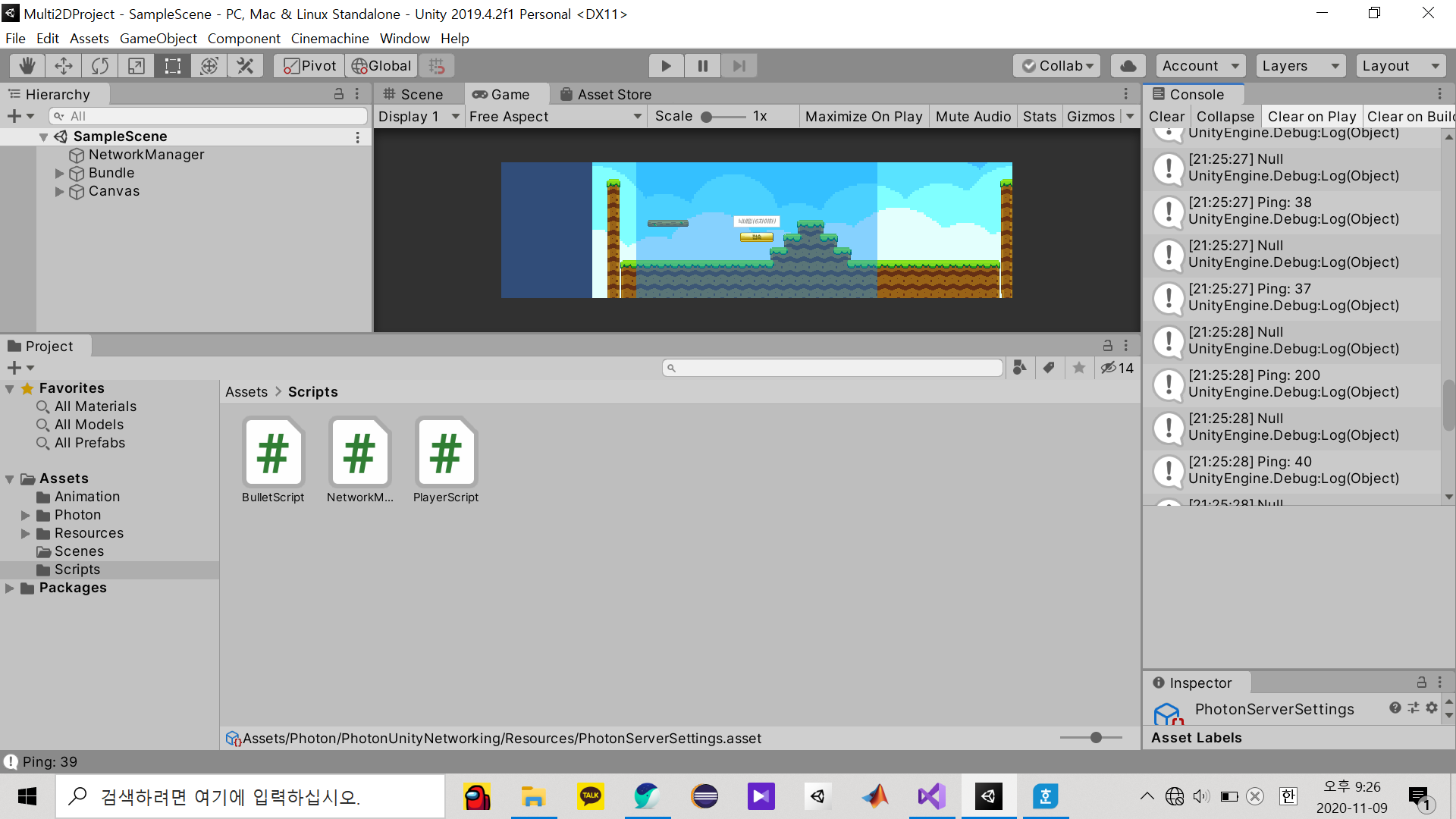
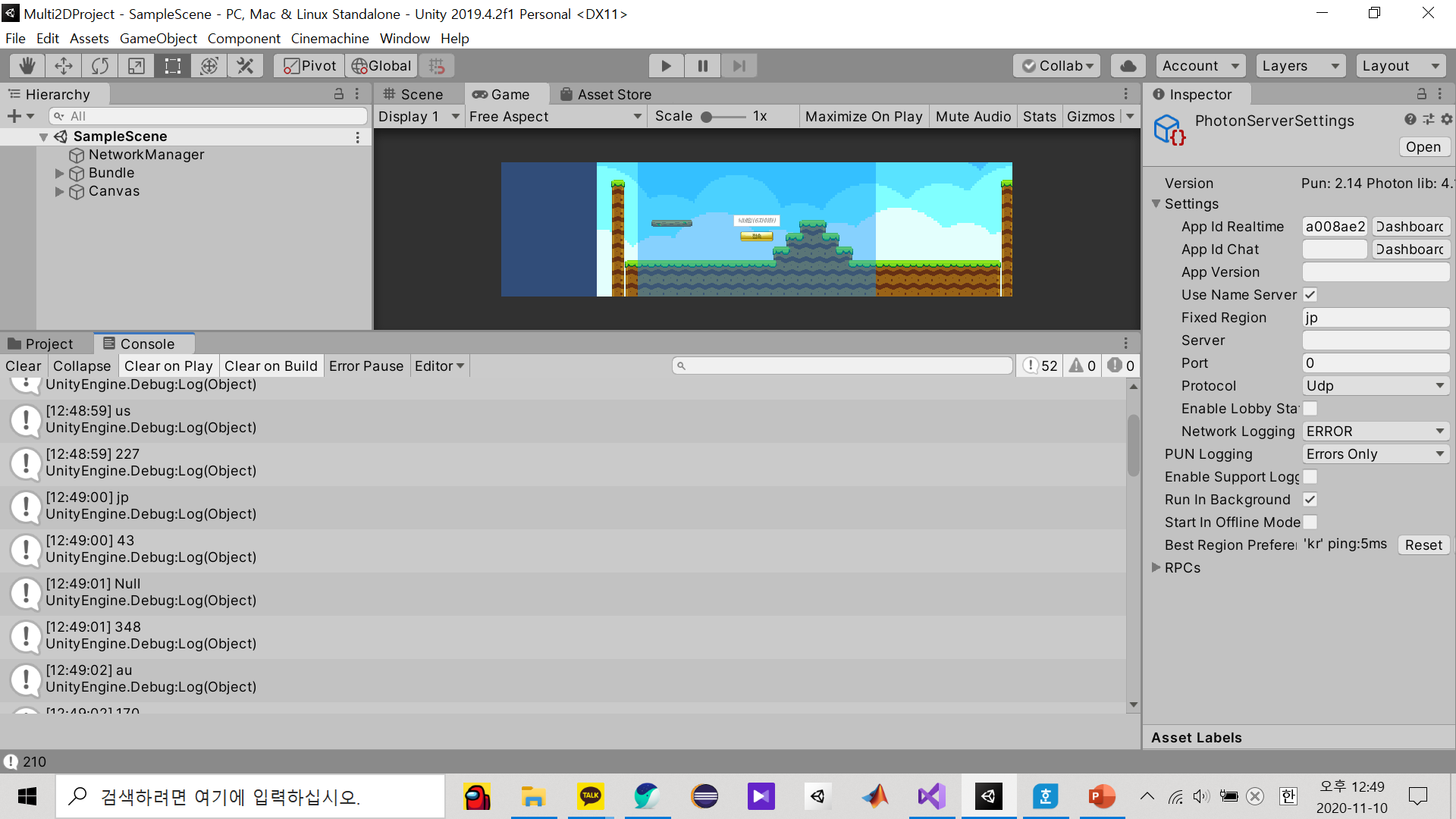
PhotonNetwork.PhotonServerSettings.AppSettings.FixedRegion = region;

실험은 2번 방식으로 진행했다.

다음은 지역 정보와 그에 대한 토큰 정보를 나열한 표이다.



InvokeRepeating()을 이용하면 정해진 시간간격마다 측정할 수 있지만, 지역 간의 편차로 인해 Null값이 나오며 ping이 제대로 측정되지 않는 것을 확인하였고, 간격을 늘려서 측정하기엔 샘플 수가 적어질 거라고 생각되었다.

<0.1~0.2초 간격으로 측정> <1초 간격으로 측정>

1초 간격으로 측정할 경우, 거리가 먼 za(남아프리카)지역에서 Null값이 발생했다.

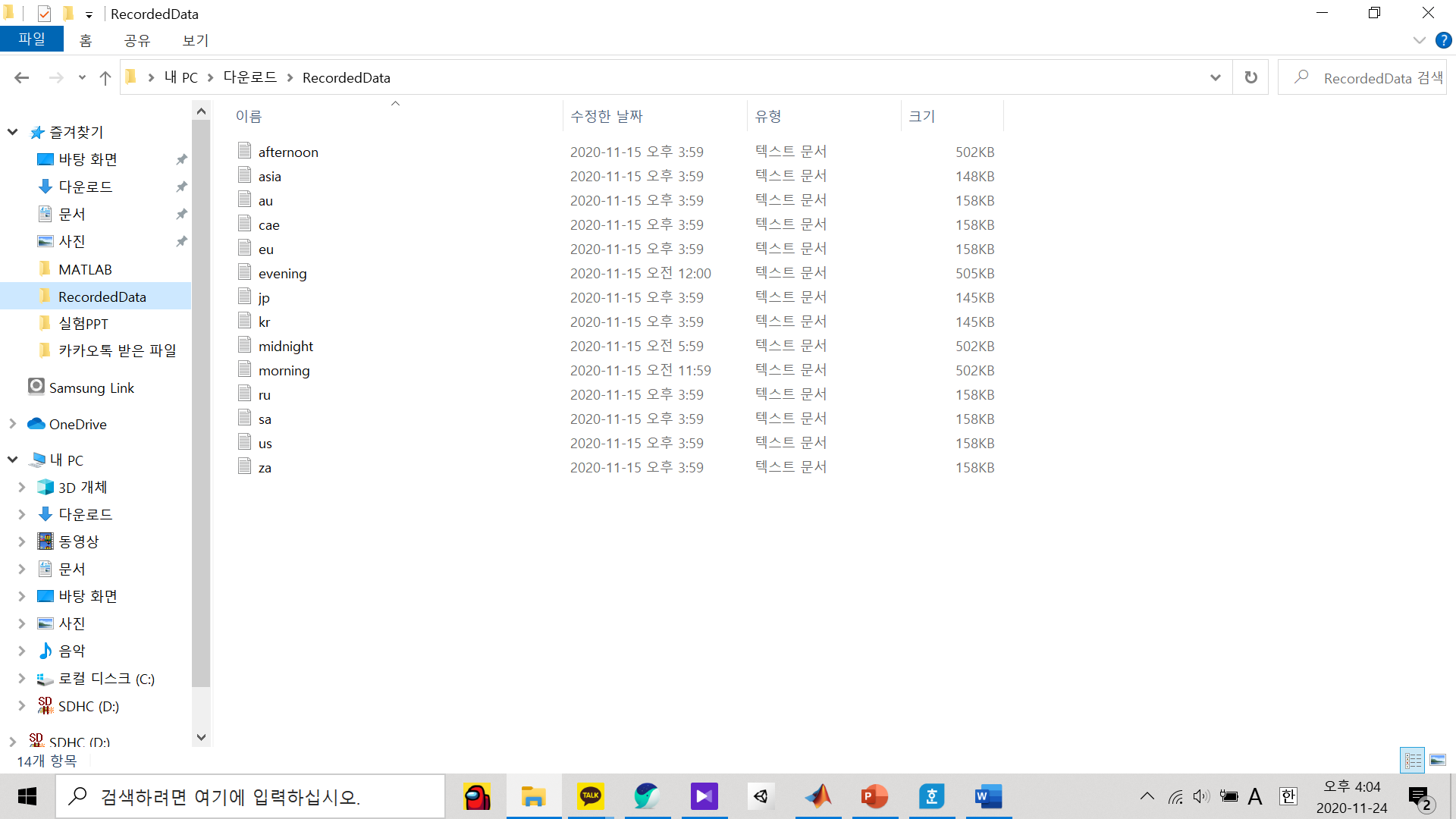
이를 위한 해결방법으로 연속적으로 호출해주는 콜백함수를 오버라이드함으로써 단계적으로 연결과 해제가 온전히 됐을 때 넘기면서, 고정시간보다 ping이 오래 걸렸을 때 수신 불량도 없애고, 시간간격을 두지않고 바로 연결되도록 했다.

실험은 10개 지역(kr, jp, asia, ru, au, cae, us, eu, sa, za)에 대해서 진행한다. 각 지역에 대한 토큰과 ping, 측정시간을 저장할 배열을 만들고, 정상적으로 측정이 완료되면 배열에 저장한다. 실험의 유효성을 위해 한 지역에서라도 Null값이 측정되거나 중간에 연결이 끊어지면 유효하지 않으므로 모든 배열을 초기화하고 다시 저장한다.

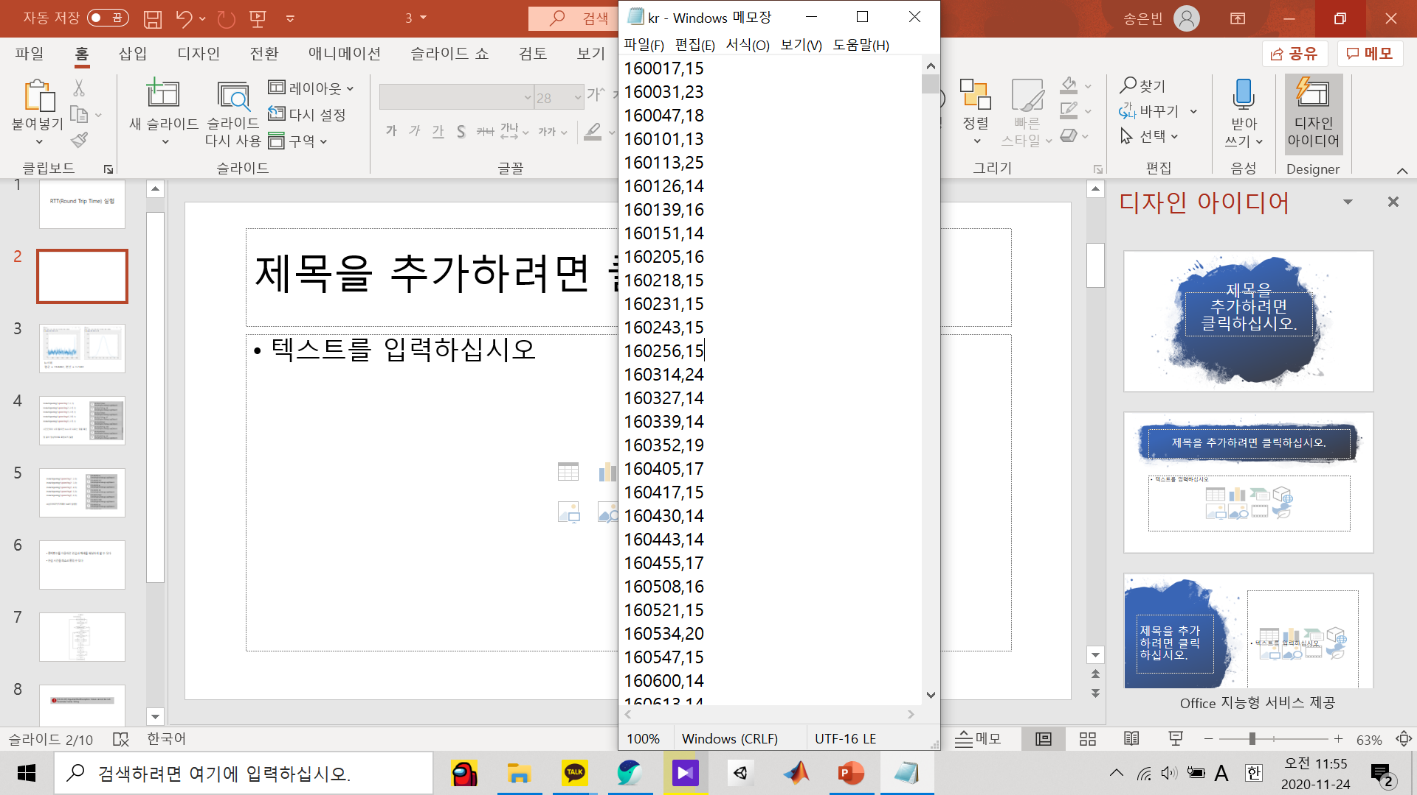
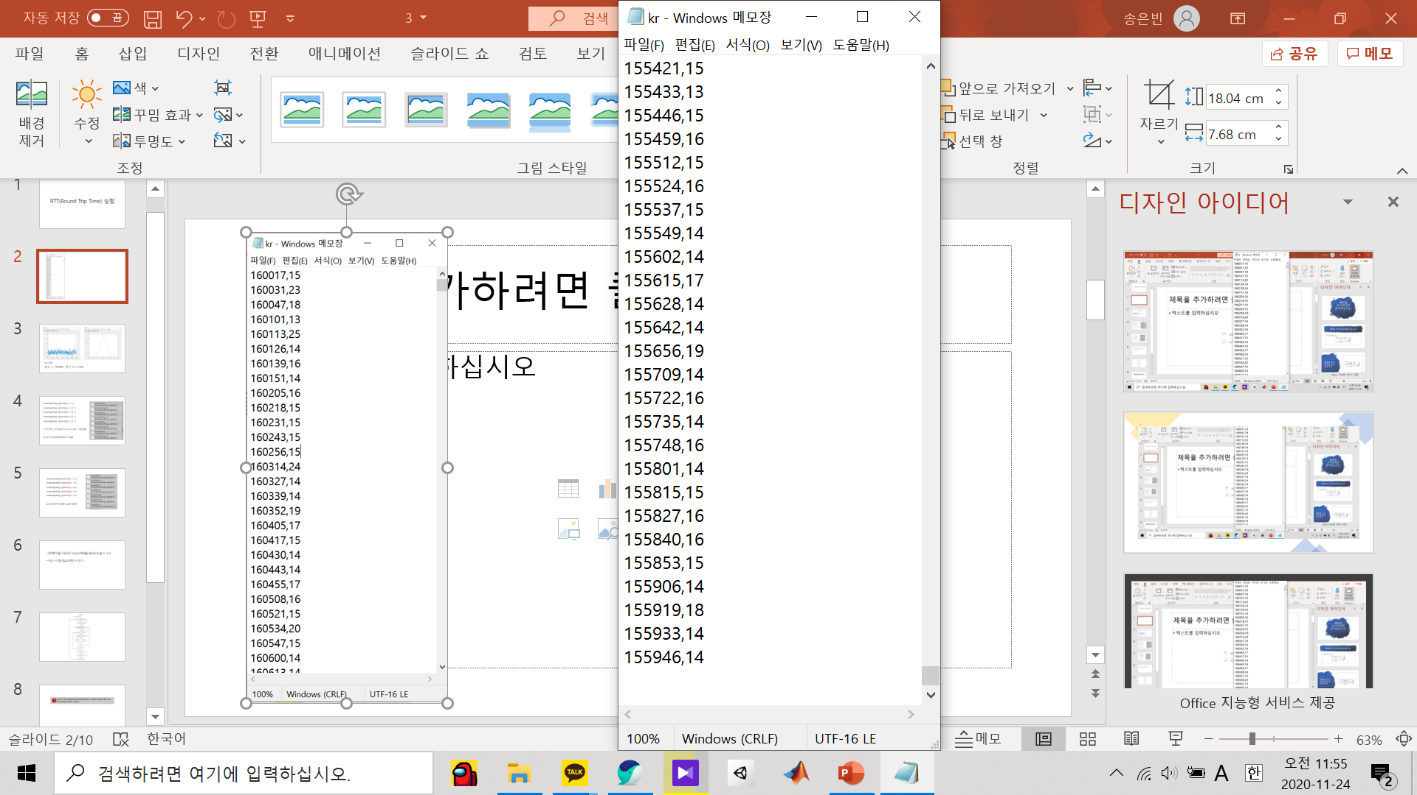
데이터 배열이 가득 찼을 때, 유효성 검사를 해서 true가 나오면 txt파일로 저장하고, false가 나오면 모든 데이터를 삭제하고 다시 저장한다.

오류 없이 10개의 지역에 대한 ping을 배열에 저장하면 filestream을 이용해 각 지역과 시간에 대한 데이터를 txt파일로 저장한다. 각각의 정보는 “,”로 구분한다.

1. 같은 지역에 시간별 차이를 보기 위해 morning(6-12), afternoon(12-18), evening(18-24), midnight(0-6)에 대한 파일 (시간, ping, 지역 출력) – 4개파일
2. 같은 시간에 지역별 차이를 보기 위해 각 지역별 모든 시간에 대한 파일 (시간, ping 출력) – 10개파일

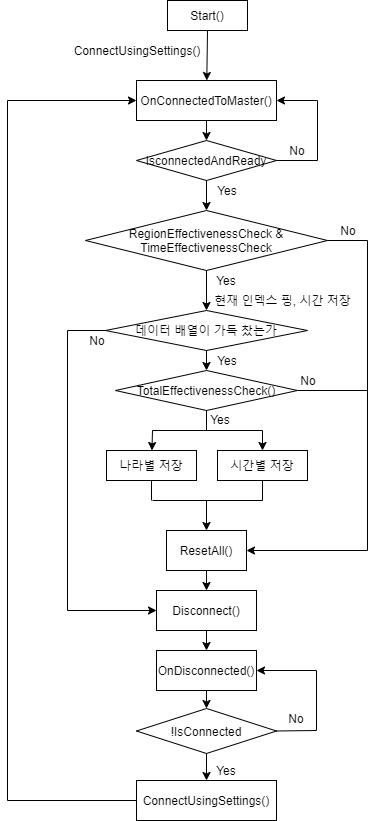


<txt 파일 캡쳐>

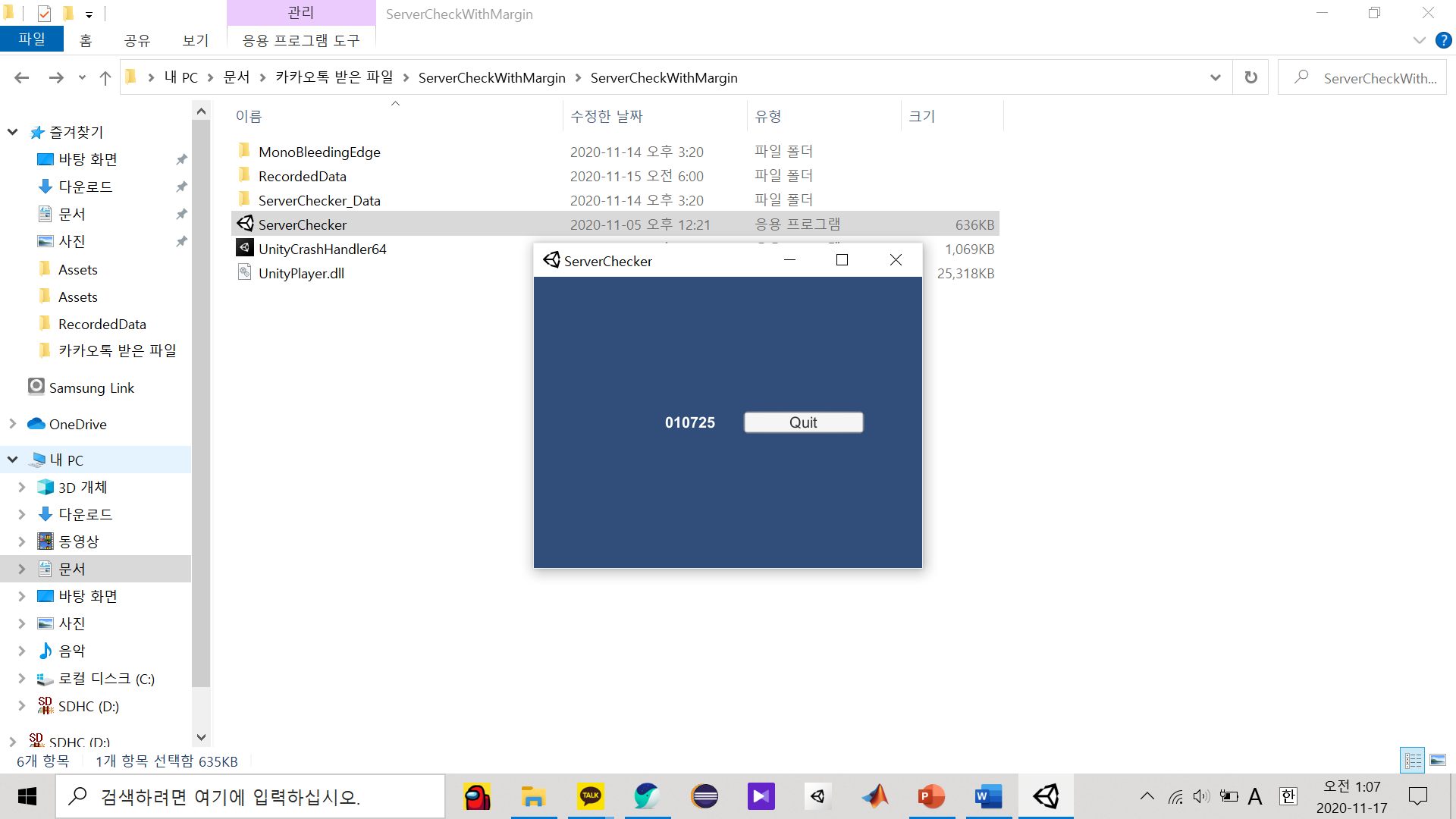
,(콤마)를 기준으로 앞에 있는 숫자는 시간(시, 분, 초)을 나타내고, 뒤에 있는 숫자는 ping 수치를 나타낸다. 실험은 16시부터 15시 59분까지 진행했고, 샘플 수는 각 나라별로 6712개씩 나왔다.

다음은 코드에 대한 다이어그램이다.



실험을 24시간동안 진행하기 위해서 컴퓨터에 무리가 가지 않도록 간단하게 UI만들고 최저사양으로 빌드했다. QuitManager클래스를 이용해서 저장 중에 프로그램을 종료하면 마지막까지 저장시키고 종료되도록 했다. UI에는 실행이 잘 되고 있는지 확인하기 위해 마지막으로 txt파일을 저장한 시간이 나타나게 하고, 종료버튼을 넣었다.

다음 사진에서 010725는 1시 7분 25초에 마지막으로 txt파일을 저장한 것을 나타낸다.



<네트워크 실험 UI>

**<데이터 분석>**

데이터 분석은 매트랩으로 진행했다.

3-1) 시간별

readmatrix을 이용해서 ,로 분리된 txt파일을 행렬로 저장한 후, 실험이 16시부터 진행됐으므로 시간을 기준으로 오름차순 정렬했다. 이를 이용하여 각 나라의 평균과 분산을 계산하고, linspace로 x축을 24로 쪼개서 그래프를 그렸다. y축은 변화폭이 크기 때문에 semilogy를 이용하여 log축으로 나타냈다.

각 나라별 데이터의 개수가 6712개이므로 for문을 6712번 실행하며 각 시간을 10000으로 나눠서 0-6(새벽), 6-12(아침), 12-18(오후), 18-24(밤)에 저장되도록 했다.

예를 들어, txt파일에 시간이 160534의 형태로 저장되므로 160534/10000=16, 즉 오후로 저장된다.

각각의 시간을 저장할 때마다 cnt를 1씩 더하면서 시간별로 샘플 수가 몇 개인지도 나타냈다.

3-2) 지역별

위에서 구한 각 나라에 대한 평균을 geobubble을 이용해서 세계지도에 나타냈다.

각 나라의 경도와 위도는 구글 지도에서 참고했다. discretize를 이용하여 평균이 0-100, 100-250, 250-400일 경우에 각각 색깔을 다르게 표시했다.

3-3) 히스토그램

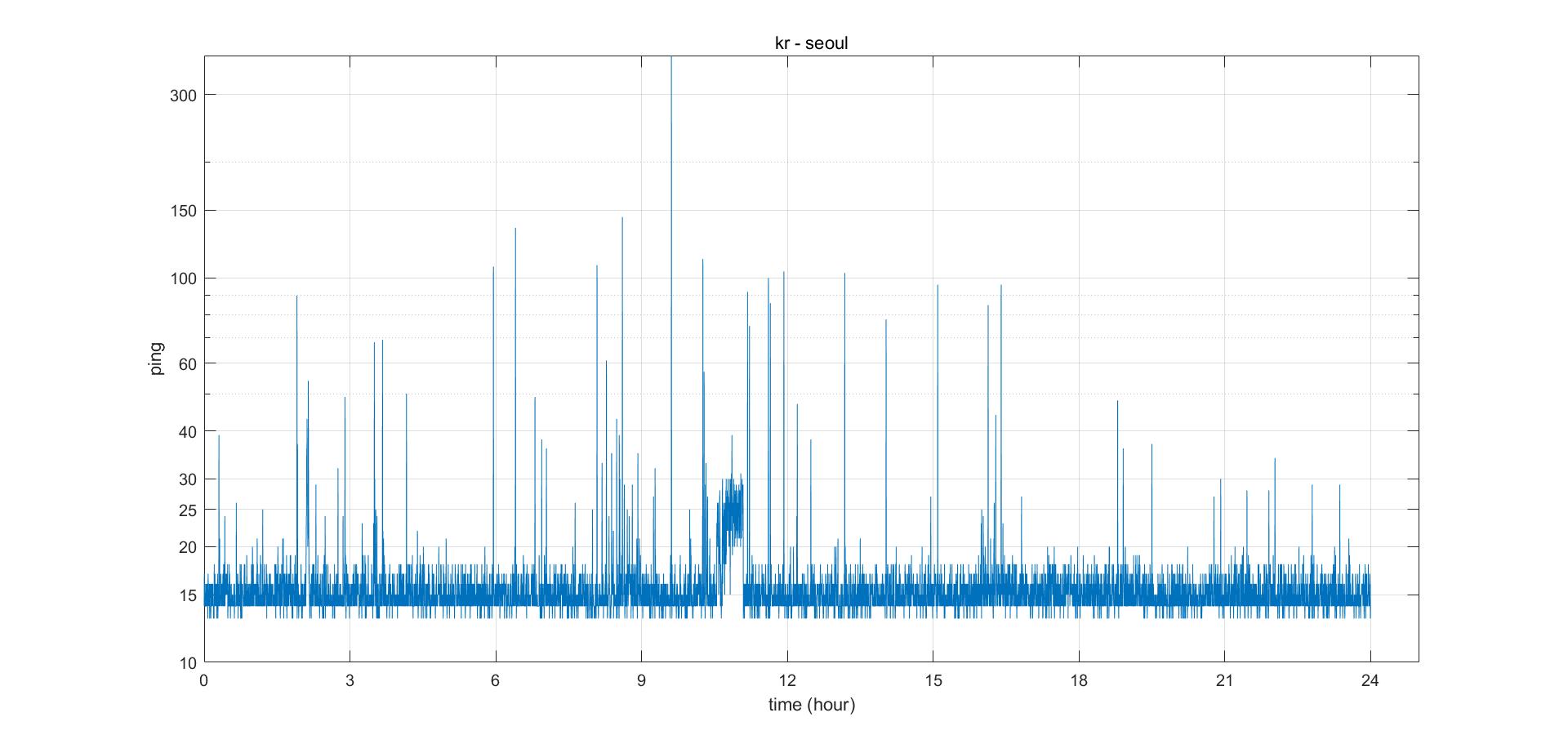
각 나라별로 신뢰도를 알아보기 위해서 히스토그램을 그렸다. 히스토그램을 보며 확연히 낮아질 때부터 지연이라고 생각하고, 그 이상되는 모든 ping 값을 마지막 값으로 대입시켰다. 나라별로 평균 ping 값이 다르므로 모두 다르게 설정했다.

**<네트워크 실험 결과>**

1. 시간에 따른 ping 속도

다음은 10개의 나라에 대한 ping을 시간별로 나타낸 그래프이다.

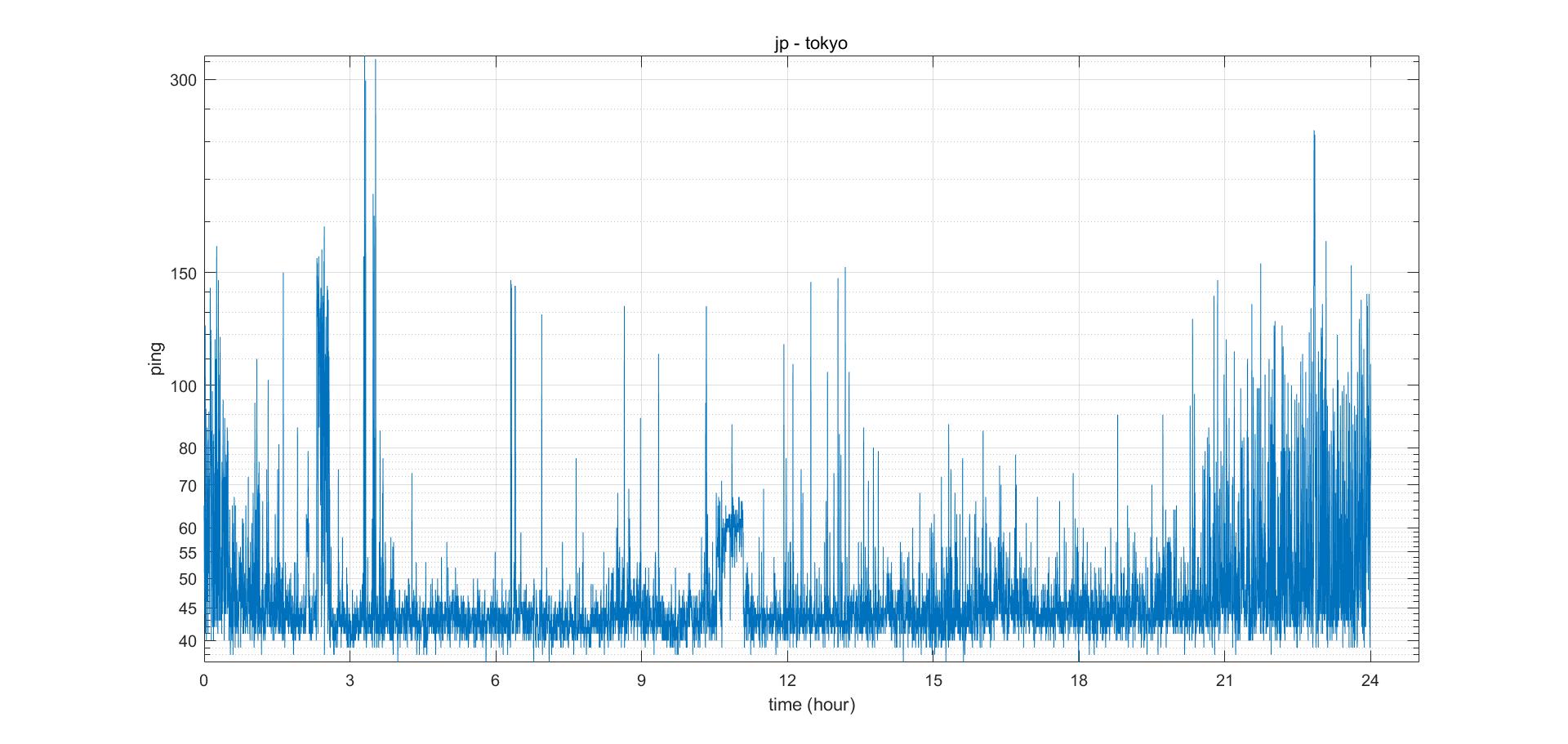
한국, 서울 (kr)



평균 ping 속도 = 15.6625ms

분산 = 46.3950

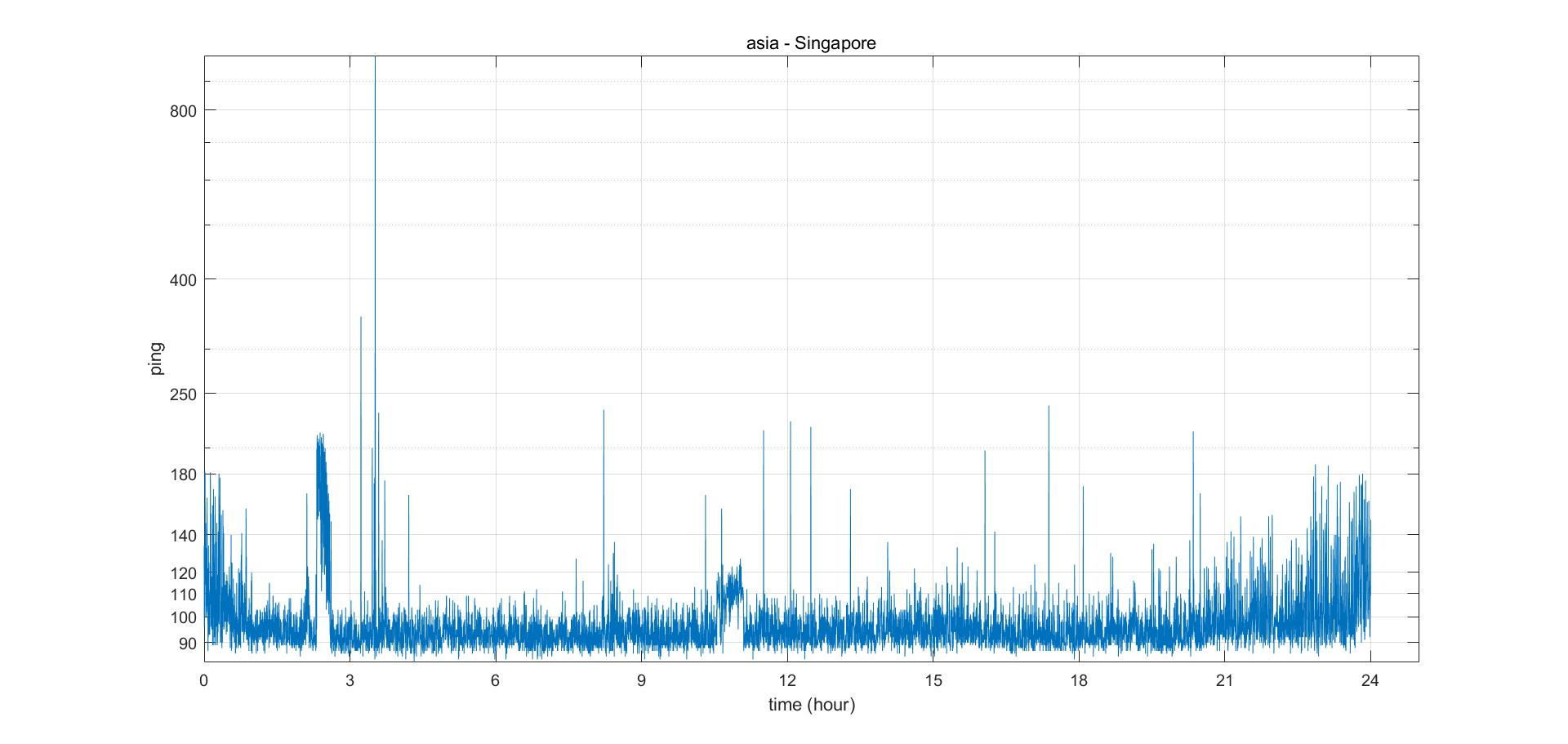
일본, 도쿄 (jp)



평균 ping 속도 = 48.2813ms

분산 = 259.0879

아시아, 싱가포르 (asia)



평균 ping 속도 = 97.6545ms

분산 = 322.0420

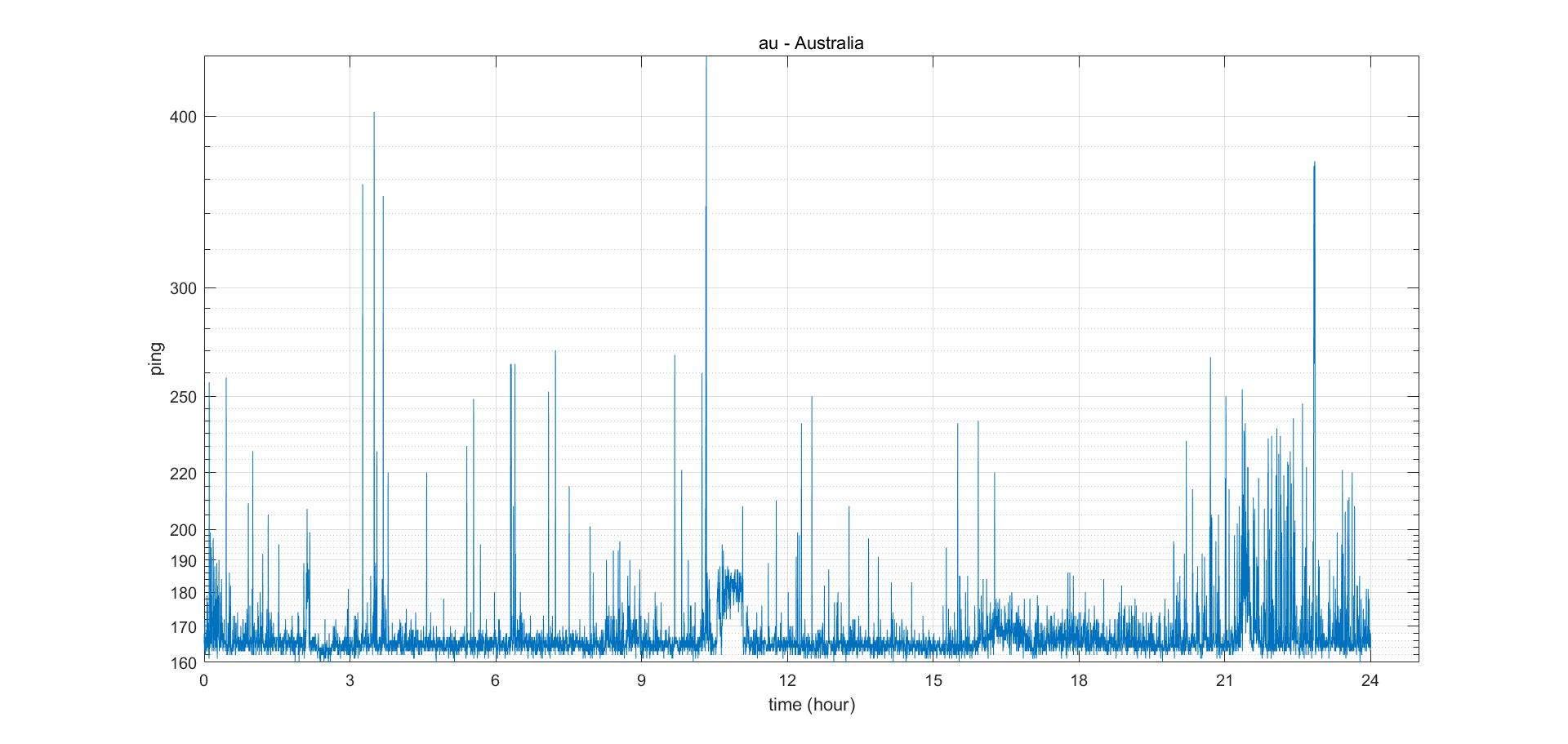
러시아, 모스코바 (ru)



평균 ping 속도 = 198.7682ms

분산 = 784.1671

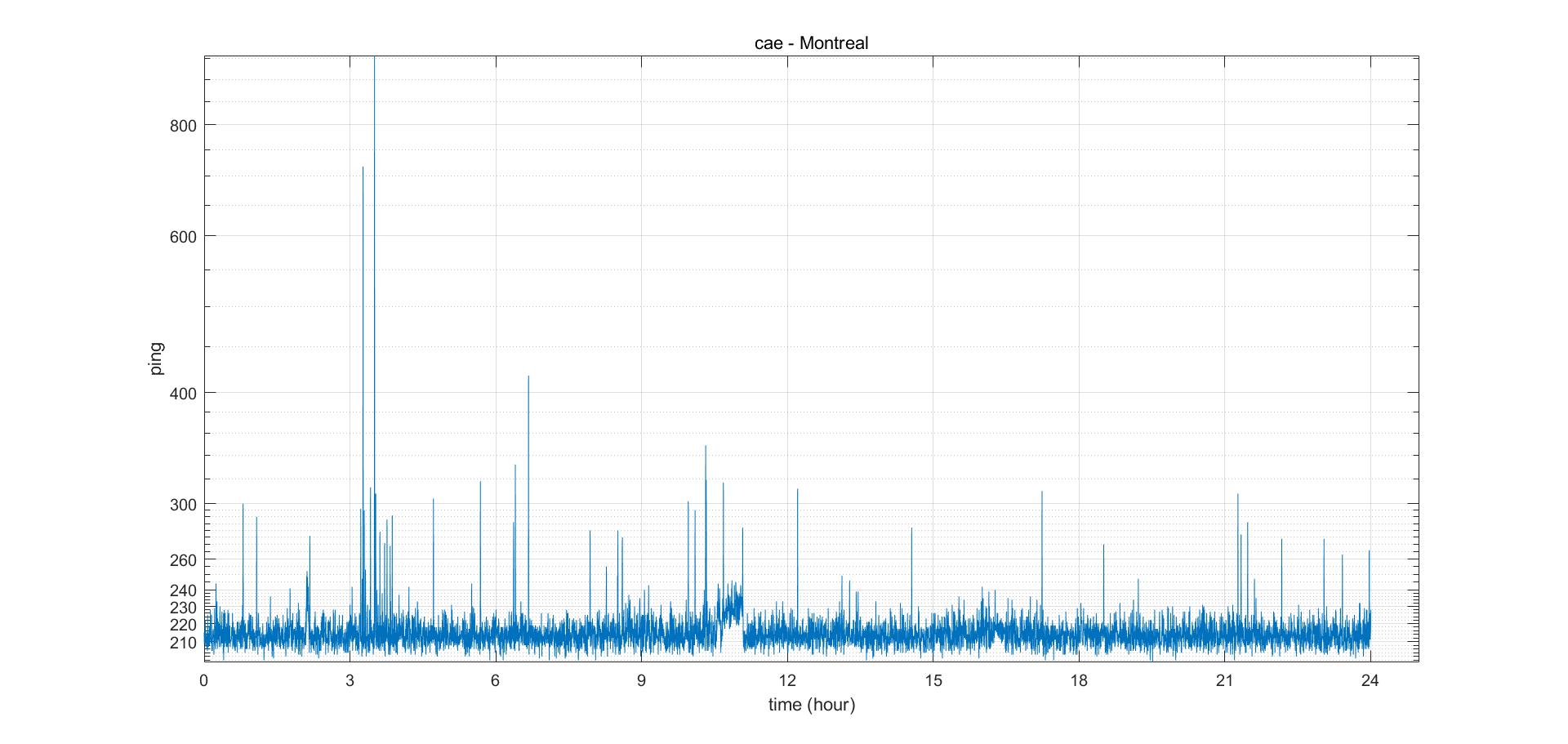
호주, 멜버른 (au)



평균 ping 속도 = 167.5513ms

분산 = 137.6481

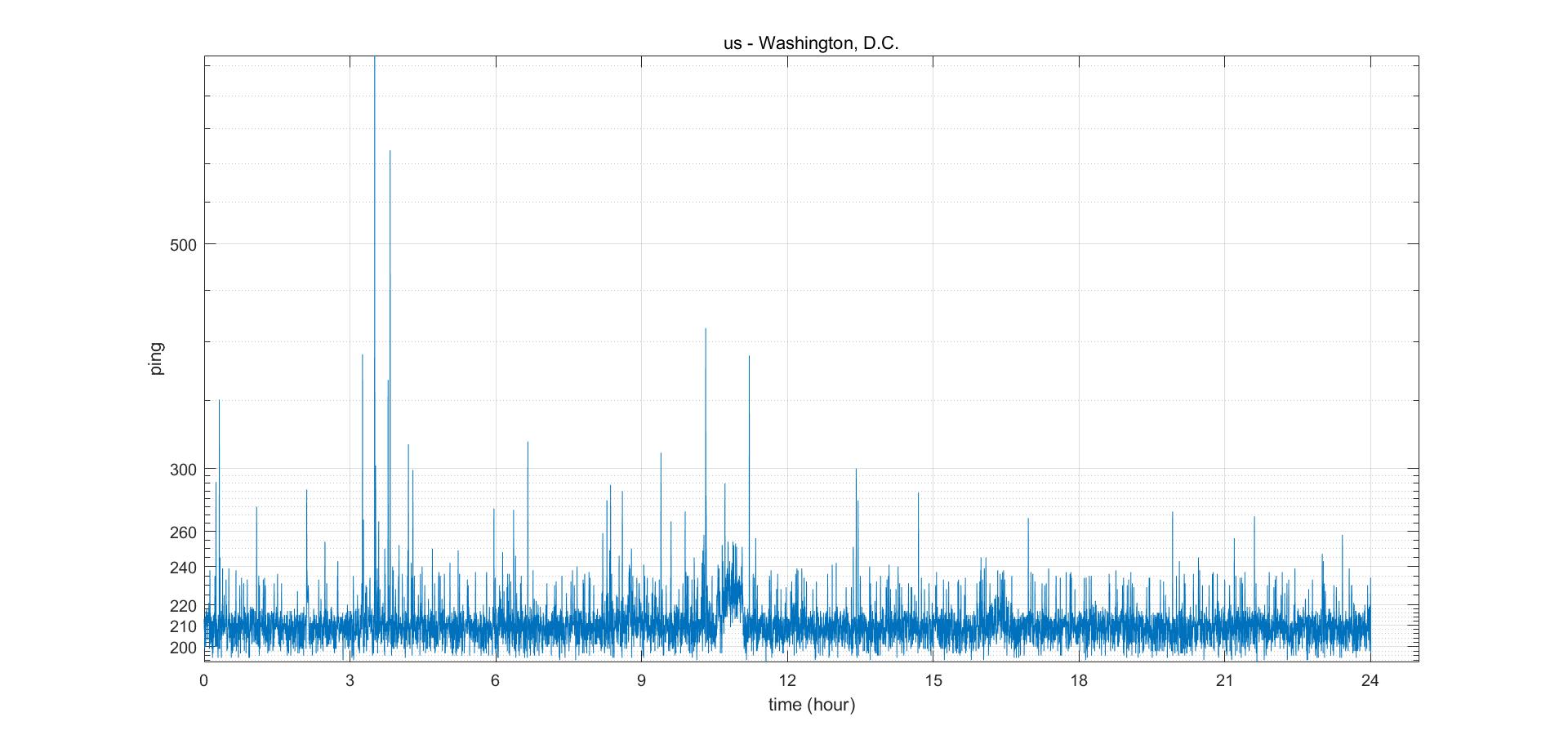
캐나다 동부, 몬트리올 (cae)



평균 ping 속도 = 214.7925ms

분산 = 206.1883

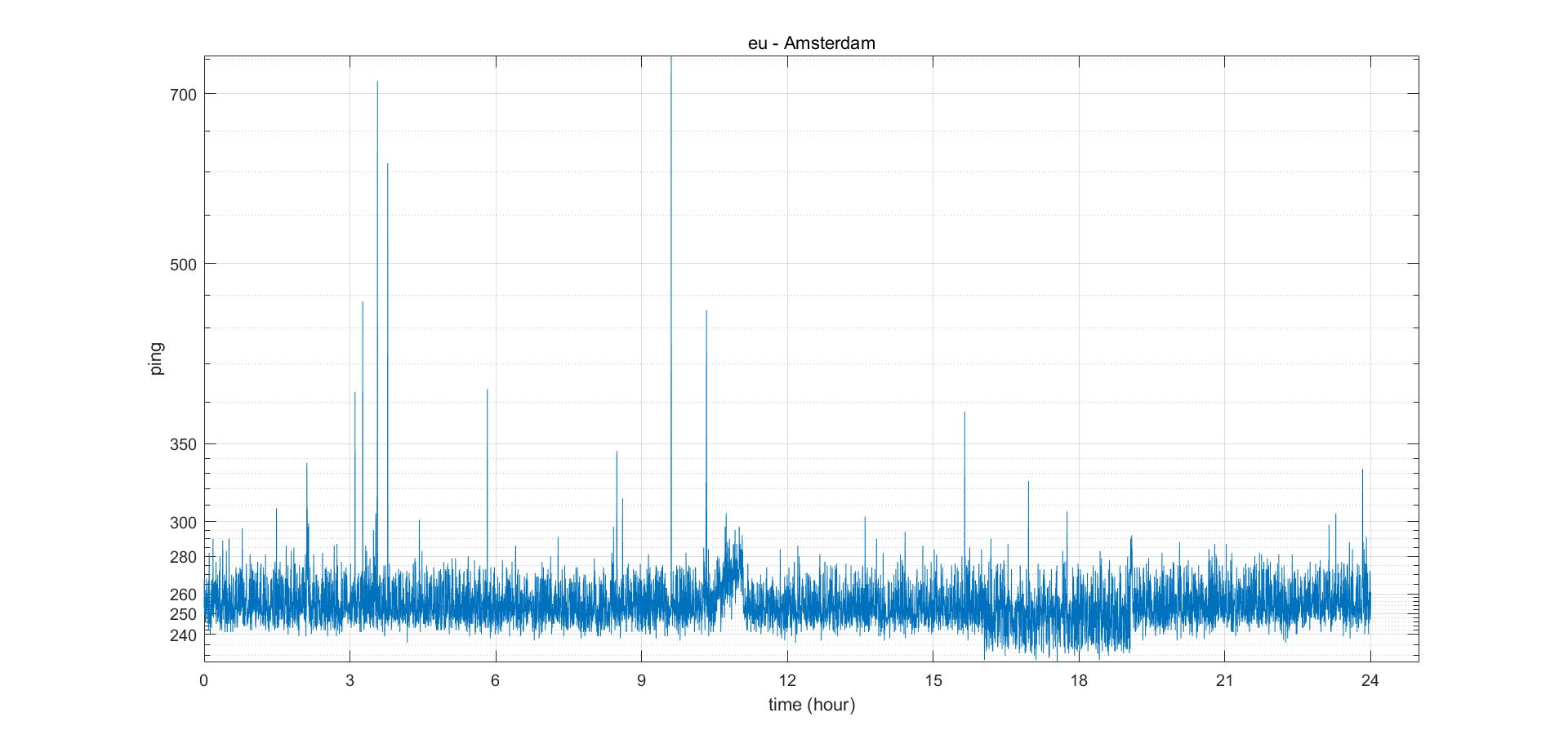
미국 동부, 워싱턴 D.C. (us)



평균 ping 속도 = 210.5693ms

분산 = 186.4002

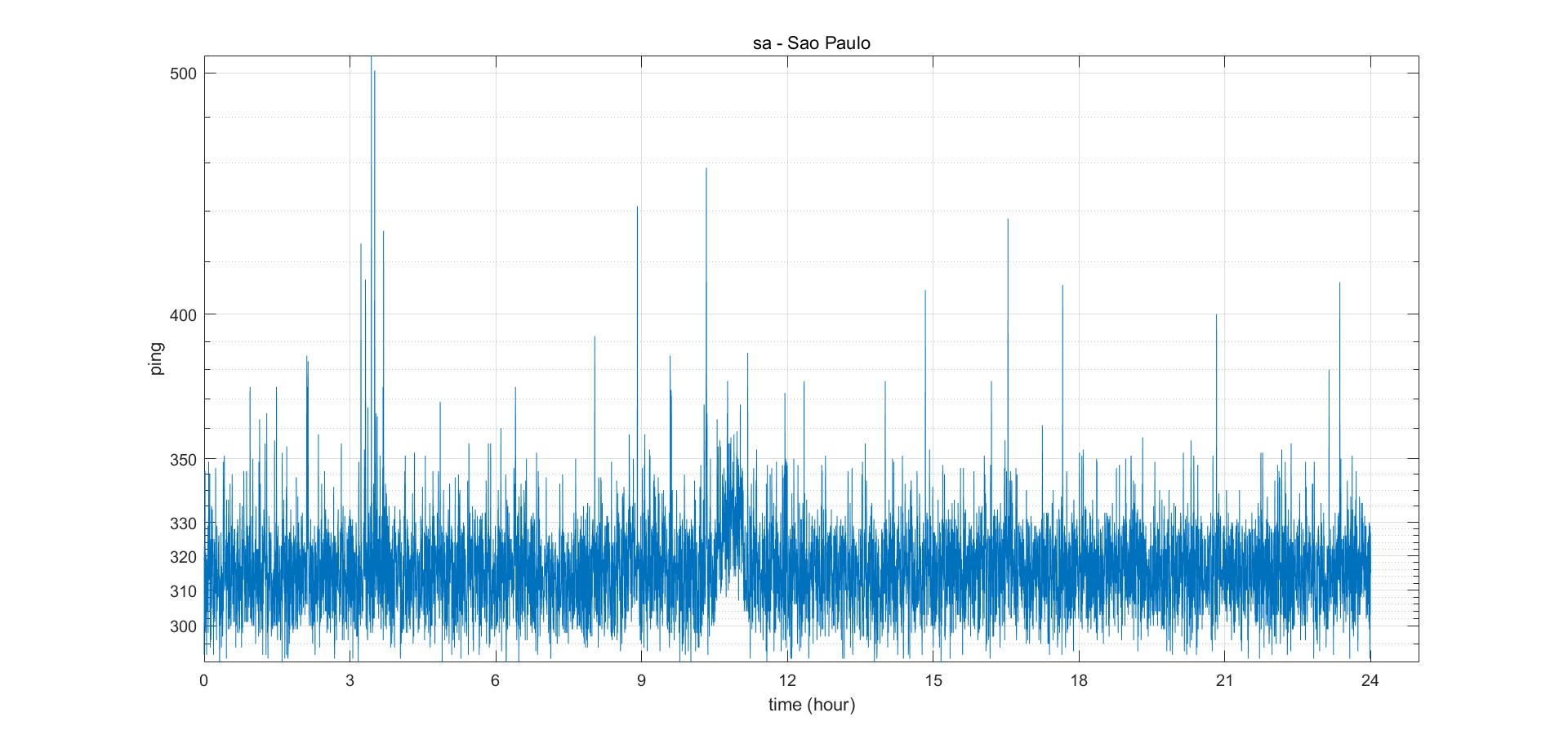
유럽, 암스테르담 (eu)



평균 ping 속도 = 254.9663ms

분산 = 219.2139

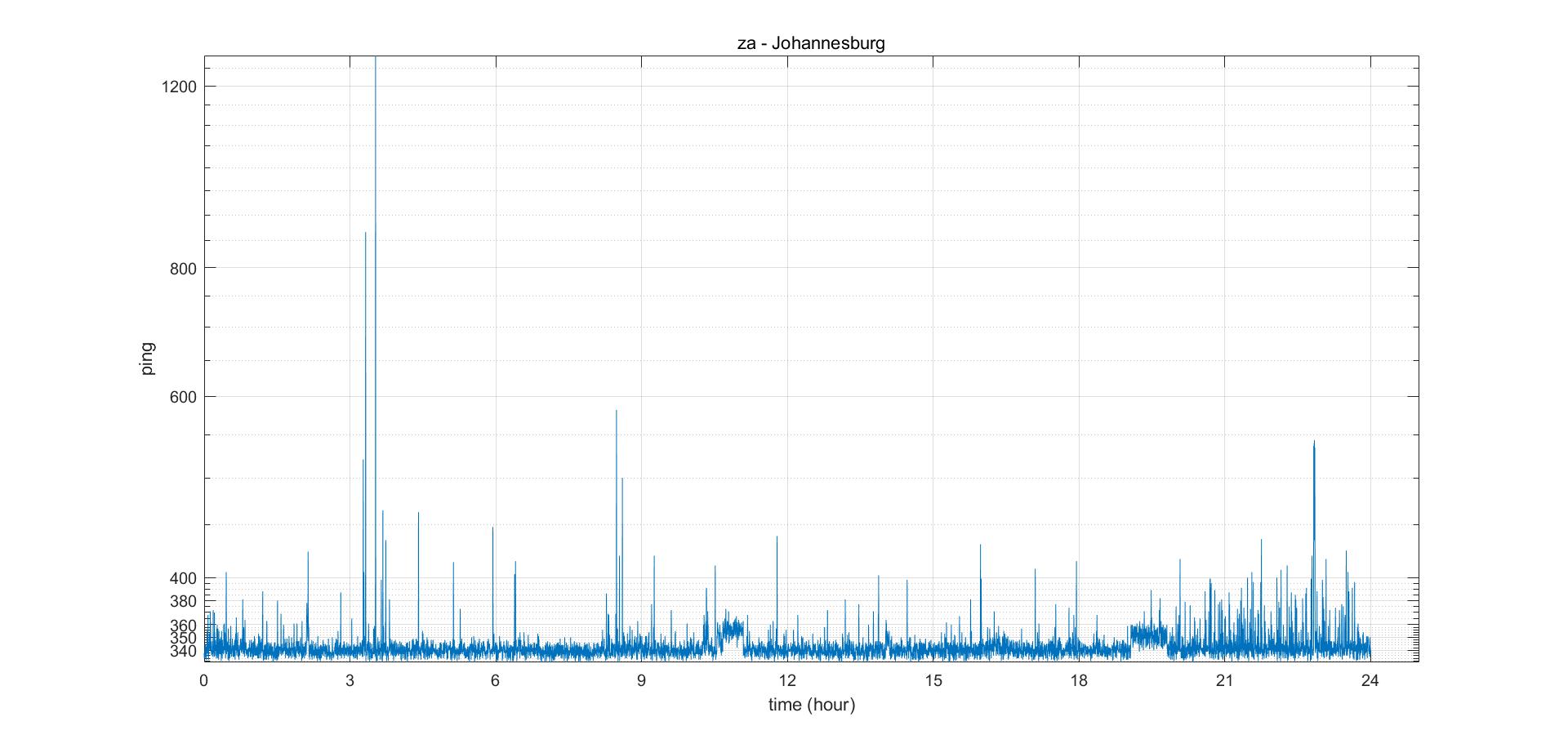
남아메리카, 상파울루 (sa)



평균 ping 속도 = 316.3619ms

분산 = 182.1615

남아프리카, 요하네스버그 (za)



평균 ping 속도 = 342.6903ms

분산 = 301.7760

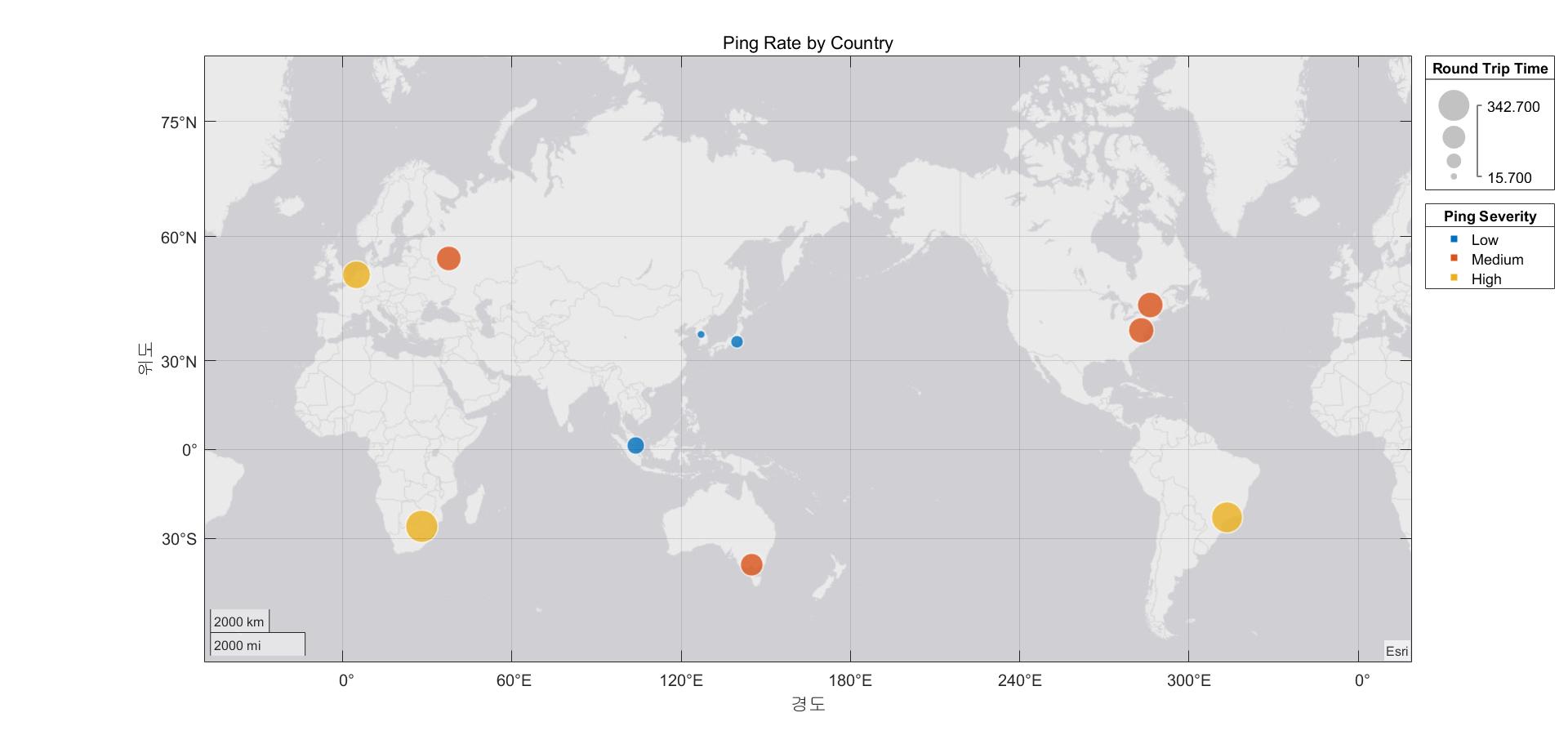
1. 시간별 평균 ping 속도

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0-6(midnight) | 6-12(morning) | 12-18(afternoon) | 18-24(evening) |
| 한국, 서울 (kr) | 15.4665 | 16.7060 | 15.3733 | 15.1063 |
| 일본, 도쿄 (jp) | 49.7222 | 45.7698 | 45.2504 | 52.3682 |
| 아시아, 싱가포르 (asia) | 100.1852 | 95.2850 | 95.0894 | 100.0529 |
| 러시아, 모스코바 (ru) | 198.5753 | 199.0107 | 196.5951 | 200.8824 |
| 호주, 멜버른 (au) | 166.5902 | 167.8801 | 166.0817 | 169.6425 |
| 캐나다 동부, 몬트리올 (cae) | 215.2372 | 216.0489 | 213.9839 | 213.9044 |
| 미국 동부, 워싱턴 D.C. (us) | 210.9946 | 212.2403 | 209.8140 | 209.2346 |
| 유럽, 암스테르담 (eu) | 256.3170 | 256.1067 | 252.7567 | 254.6875 |
| 남아메리카, 상파울루 (sa) | 315.9582 | 317.4705 | 315.9618 | 316.0576 |
| 남아프리카, 요하네스버그 (za) | 342.3409 | 342.2838 | 340.9261 | 345.2008 |

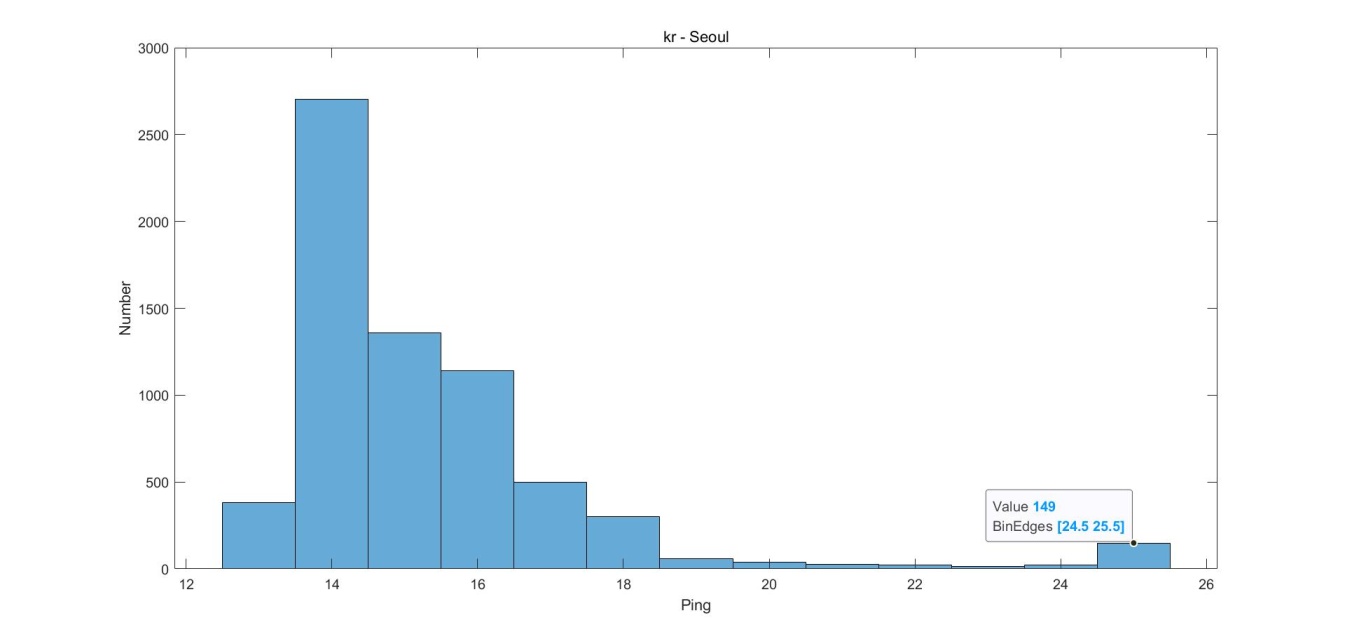
매트랩의 geobubble을 이용해서 평균 ping을 위도와 경도를 기준으로 하여 세계지도에 나타냈다. 지역별 위도와 경도는 아래 표와 같이 설정했다.

|  |  |  |
| --- | --- | --- |
| 지역 | 위도 | 경도 |
| 한국, 서울 (kr) | 37.7 | 127 |
| 일본, 도쿄 (jp) | 35.7 | 139.8 |
| 아시아, 싱가포르 (asia) | 1.4 | 103.9 |
| 러시아, 모스코바 (ru) | 55.8 | 37.6 |
| 호주, 멜버른 (au) | -37.8 | 145 |
| 캐나다 동부, 몬트리올 (cae) | 45.5 | -73.6 |
| 미국 동부, 워싱턴 D.C. (us) | 38.9 | -77 |
| 유럽, 암스테르담 (eu) | 52.4 | 4.9 |
| 남아메리카, 상파울루 (sa) | -23.5 | -46.6 |
| 남아프리카, 요하네스버그 (za) | -26.2 | 28 |

0-100ms 파랑, 100-250ms 빨강, 250-400ms 노랑

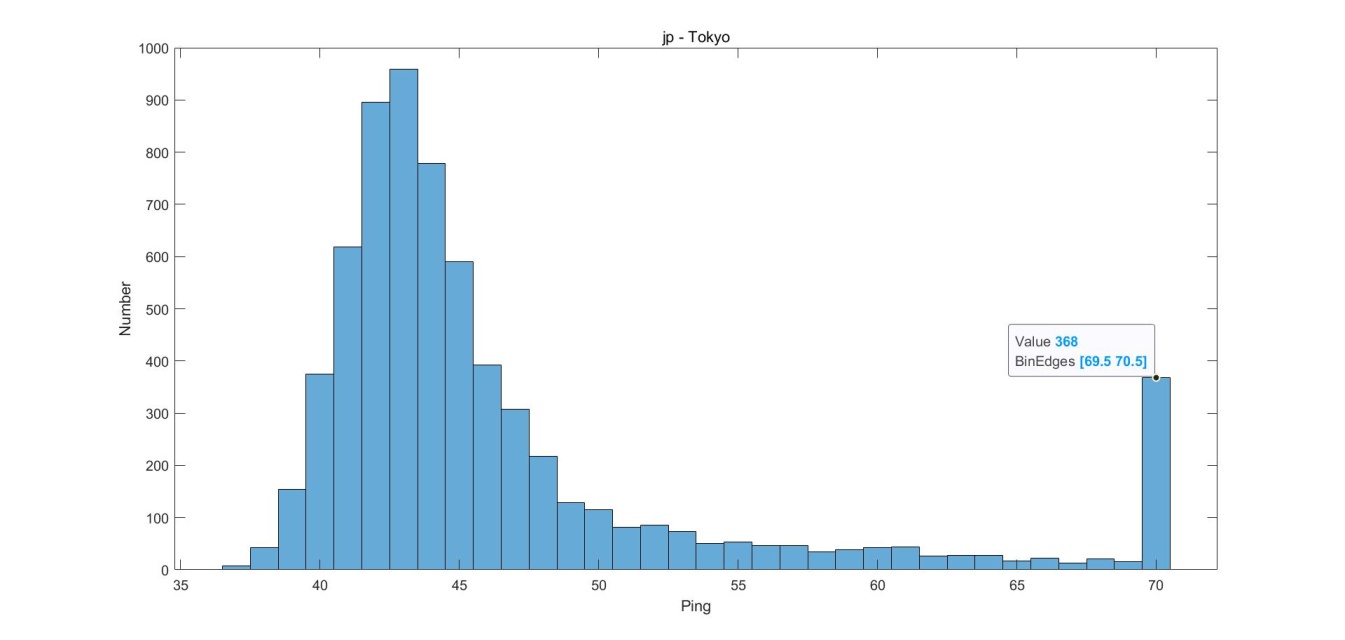


1. 각 나라별 지연율



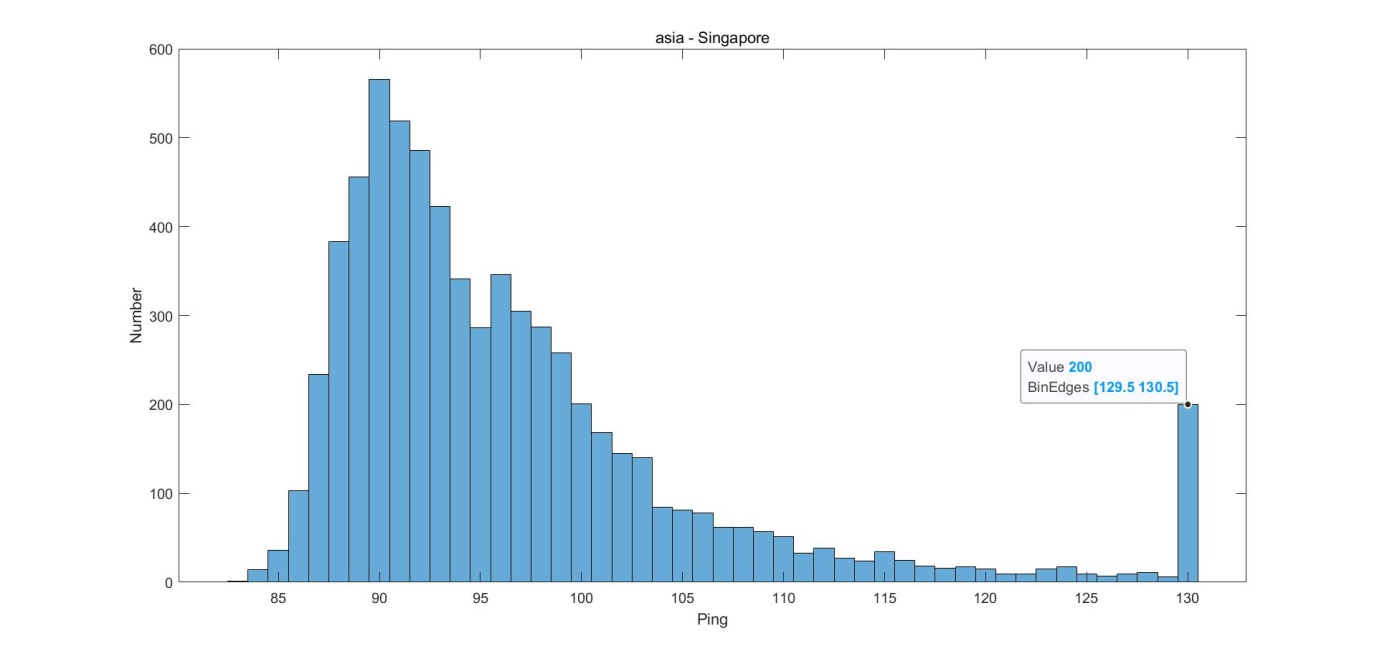
149/6712\*100 = 2.21%

일본, 도쿄 (jp)



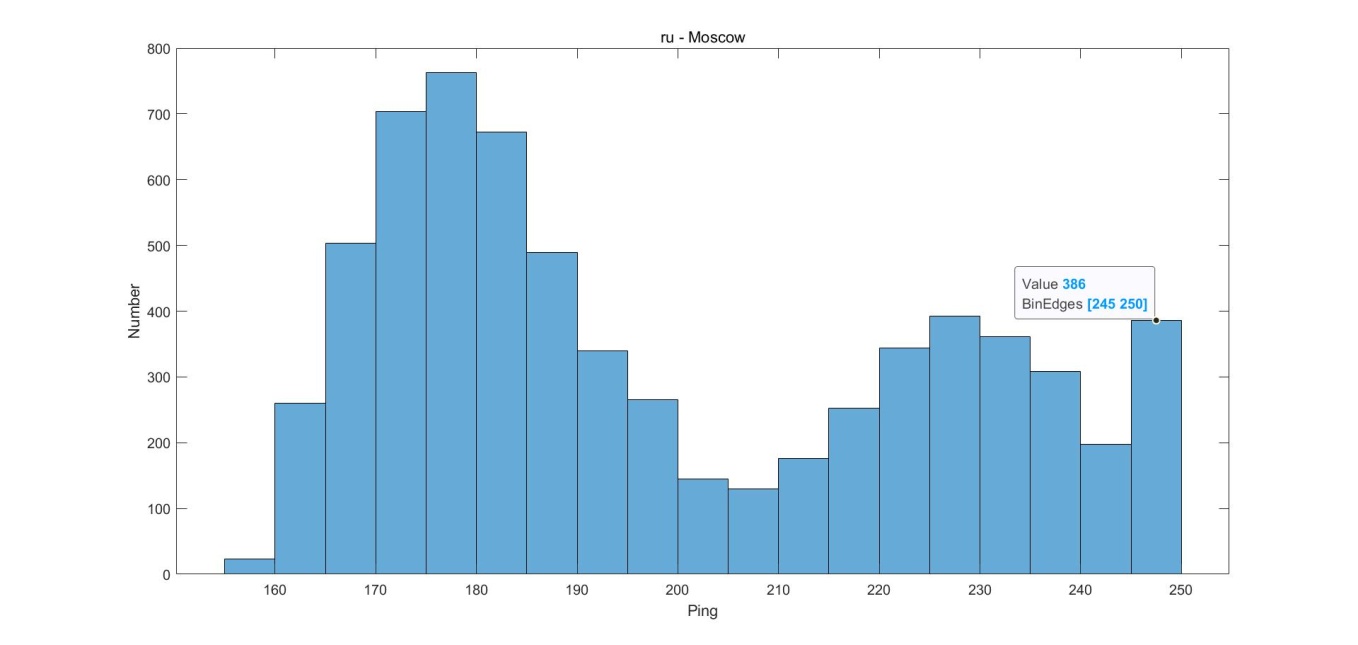
158/6712\*100 = 5.48%

아시아, 싱가포르 (asia)



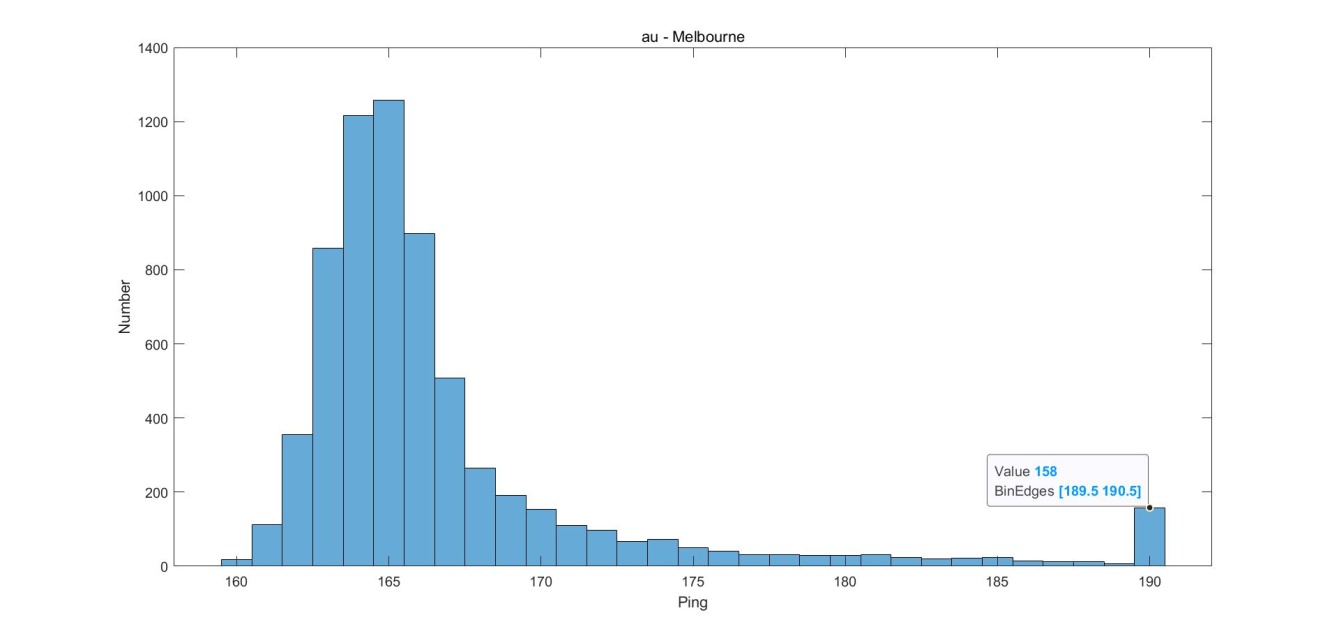
200/6712\*100 = 2.97%

러시아, 모스코바 (ru)



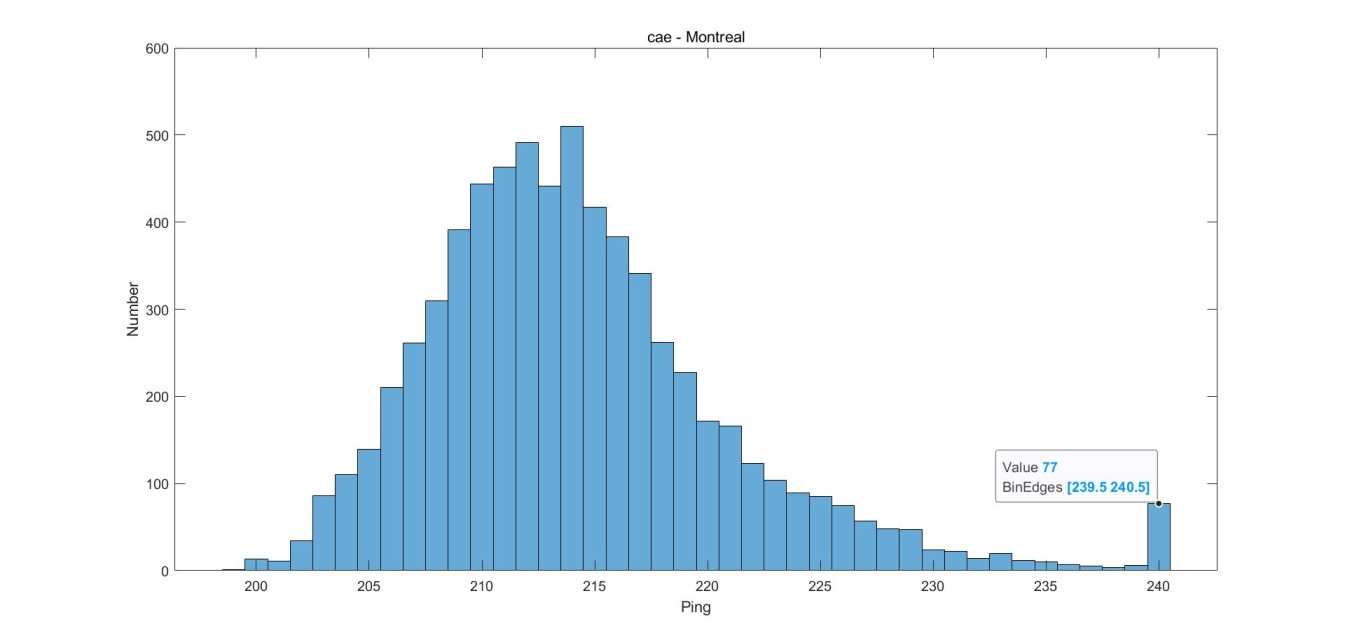
386/6712\*100 = 5.75%

호주, 멜버른 (au)



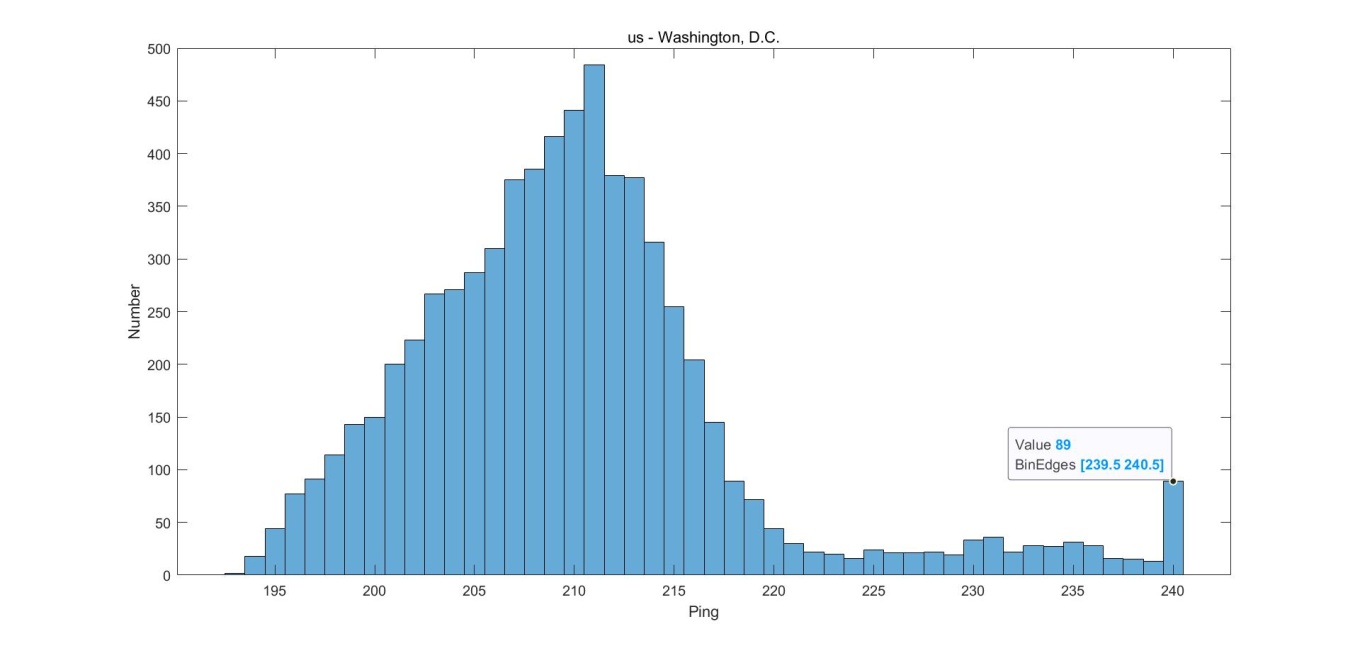
158/6712\*100 = 2.35%

캐나다 동부, 몬트리올 (cae)



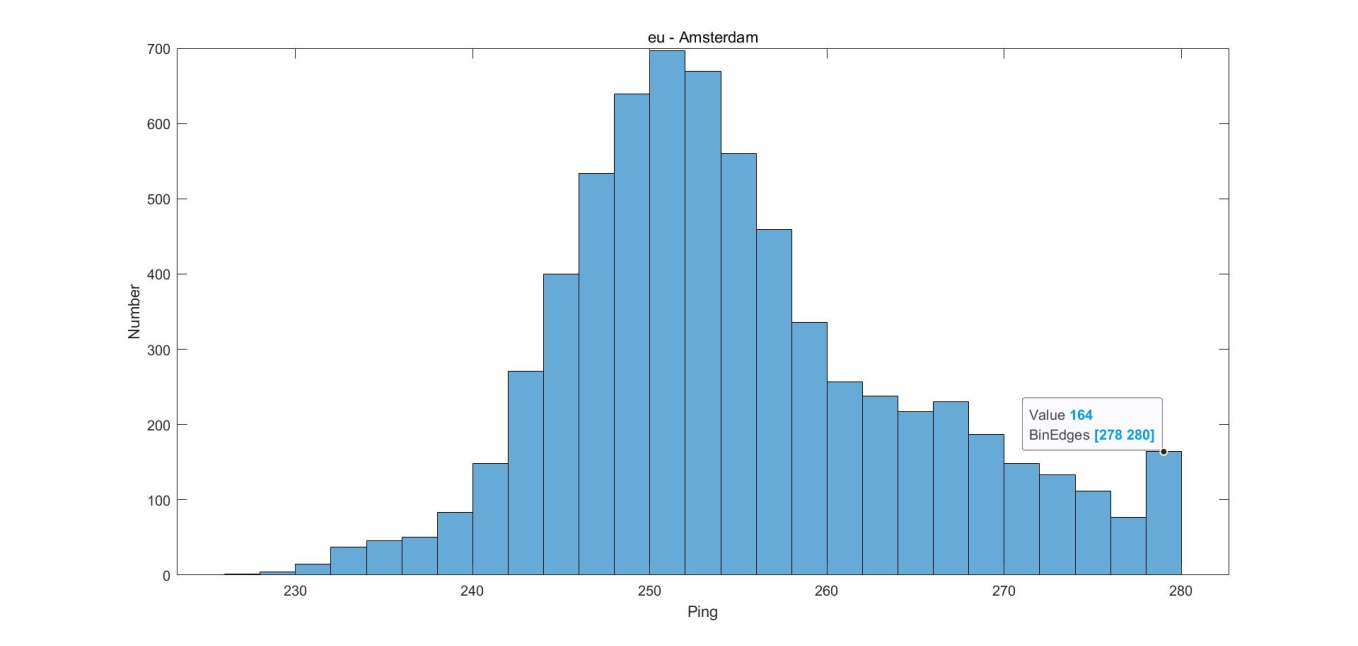
77/6712\*100 = 1.14%

미국 동부, 워싱턴 D.C. (us)



89/6712\*100 = 1.32%

유럽, 암스테르담 (eu)



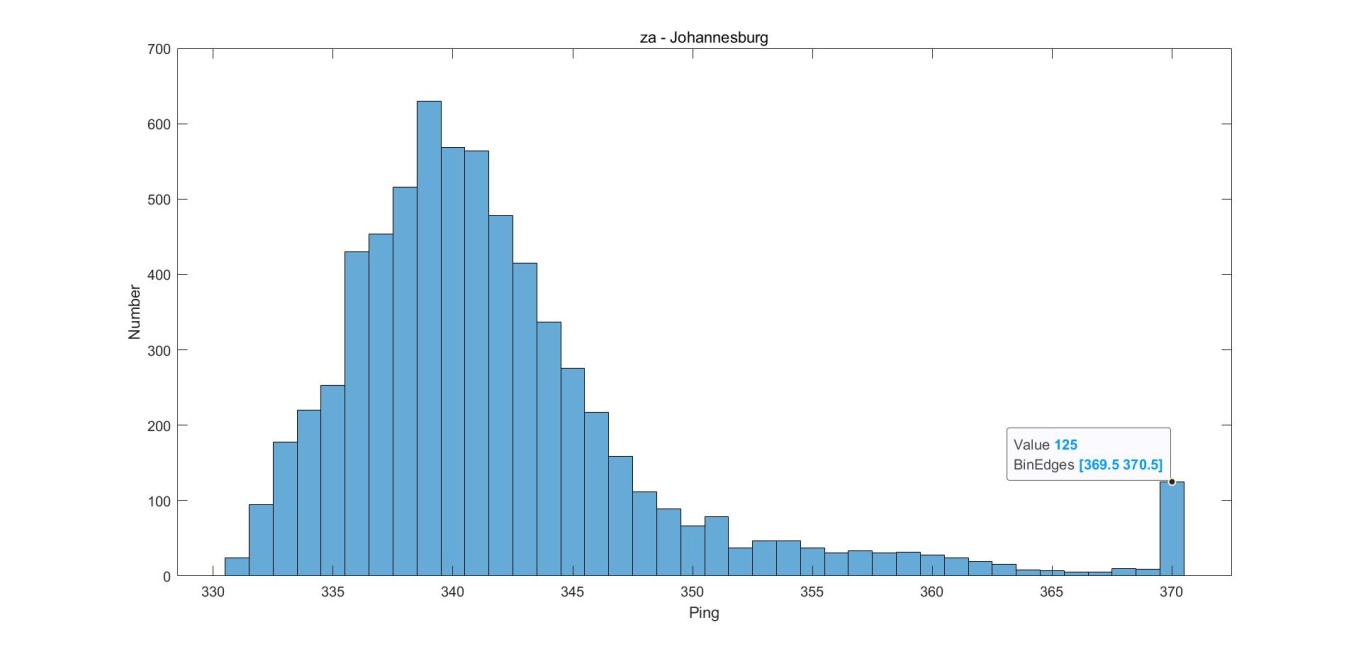
164/6712\*100 = 2.44%

남아메리카, 상파울루 (sa)



144/6712\*100 = 2.14%

남아프리카, 요하네스버그 (za)



125/6712\*100 = 1.86%

**<실험 결과 분석>**

포톤 클라우드 서버에서 10개의 지역에 서버를 두고 ping을 보냈을 때, 서버에 도달했다가 다시 되돌아오기까지의 시간(RTT, Round Trip Time)에 대한 측정을 진행했다. 각 지역에서 시간에 따른 차이와 각 지역별 차이, 각 지역별 지연율을 나타냈다.

시간에 따른 ping의 분포를 봤을 때, 예상했던 것과 다르게 시간별로 큰 차이는 없었다. 이를 정확한 수치로 보기 위해 각 시간별 평균을 나타냈고, 큰 차이 없이 모든 시간대에 비슷하게 나타났다. 지역에 따라 0-6시, 6-12시, 18-24시에 가장 높은 수치가 나왔다.

시간별로 측정된 샘플 수를 세어본 결과, midcnt = 1674 (0-6), morcnt = 1677 (6-12), aftcnt = 1677 (12-18), evecnt = 1684 (18-24)로 0-6시에 샘플 수가 가장 적게 나왔다. 포톤 네트워크의 연결이 저장하는 도중에 끊어지거나 null값이 저장될 때, 모든 값을 리셋 시키고 다시 저장하게 했으므로 0-6시에 네트워크 상태가 가장 좋지 않았다고 볼 수 있다.

평균이 0-100인 지역(서울, 도쿄, 싱가포르)은 파란색, 100-250인 지역(모스크바, 멜버른, 몬트리올, 워싱턴 D.C.)은 빨간색, 250-400인 지역(암스테르담, 요하네스버그, 상파울루)은 노란색으로 나타나게 한 결과, 한국(서울)을 기준으로 거리가 멀어질수록 ping이 높게 측정된 것을 시각적으로 확인할 수 있었다.

각 지역에 따른 지연율은 평균 ping이 높은 지역도 있고, 낮은 지역도 있기 때문에 정확한 기준을 잡기가 애매하므로 그래프가 확연히 낮아질 때부터 지연이라고 생각하고 확률을 계산했다. 각 지역별로 1%에서 5%까지 나왔다. 이는 평균에서도 크게 떨어지지 않은 수치로 잡았으므로 완전한 지연은 그 이하로 나올 것으로 예상된다. 확률이 높게 나온 jp(도쿄)와 ru(모스크바)에서는 분산이 크기 때문에 높게 나온 것으로 생각된다.

<네트워크>

* 유니티 코드

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using Photon.Pun;

using Photon.Realtime;

using UnityEngine.UI;

using System.IO;

using System;

public class QuitManager

{

private Boolean savePoint;

private Boolean quitRequest;

public QuitManager()

{

savePoint = false;

quitRequest = false;

}

public void Quit()

{

if (savePoint) quitRequest = true;

else Application.Quit();

}

public void OnSaving()

{

savePoint = true;

}

public void OnSaved()

{

savePoint = false;

if (quitRequest) Quit();

}

}

public class NetworkManager : MonoBehaviourPunCallbacks

{

string[] regions = new string[10] { "kr", "jp", "asia", "ru", "au", "cae", "us", "eu", "sa", "za" };

int[] datas = new int[10] { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 };

string[] times = new string[10] { null, null, null, null, null, null, null, null, null, null };

private int index = -1;

public Text progress;

QuitManager quitManager;

private void Start()

{

quitManager = new QuitManager();

ResetIndex();

PhotonNetwork.PhotonServerSettings.AppSettings.FixedRegion = regions[GetNextIndex()];

PhotonNetwork.ConnectUsingSettings();

}

public void OnApplicationQuit()

{

quitManager.Quit();

}

public void OnQuit()

{

quitManager.Quit();

}

public override void OnConnectedToMaster()

{

if (PhotonNetwork.IsConnectedAndReady)

{

Writer(DateTime.Now.ToString("HHmmss"), PhotonNetwork.GetPing(), PhotonNetwork.CloudRegion);

PhotonNetwork.Disconnect();

}

}

public override void OnDisconnected(DisconnectCause cause)

{

if (!PhotonNetwork.IsConnected)

{

PhotonNetwork.PhotonServerSettings.AppSettings.FixedRegion = regions[GetNextIndex()];

PhotonNetwork.ConnectUsingSettings();

}

}

private void Writer(string time, int data, string region)

{

if (RegionEffectivenessCheck(region) && TimeEffectivenessCheck(time))

{

datas[GetNowIndex()] = data;

times[GetNowIndex()] = time;

if (GetNowIndex() == (regions.Length - 1))

{

WriteData();

ResetAll();

}

}

else

{

progress.text = "failed";

ResetAll();

}

}

private void WriteData()

{

quitManager.OnSaving();

if (TotalEffectivenessCheck())

{

FileStream timeFile = new FileStream("RecordedData/" + getTitleTime(times[0].Substring(0, 2)) + ".txt", FileMode.Append, FileAccess.Write);

StreamWriter writerTime = new StreamWriter(timeFile, System.Text.Encoding.Unicode);

writerTime.WriteLine('$');

for (int i = 0; i < regions.Length; ++i)

{

writerTime.Write(times[i]);

writerTime.Write(",");

writerTime.Write(datas[i]);

writerTime.Write(",");

writerTime.WriteLine(regions[i]);

WriteTitle(i);

}

writerTime.Close();

progress.text = times[0];

}

else progress.text = "failed";

quitManager.OnSaved();

}

private void WriteTitle(int titleIndex)

{

FileStream titleFile = new FileStream("RecordedData/" + regions[titleIndex] + ".txt", FileMode.Append, FileAccess.Write);

StreamWriter writerTitle= new StreamWriter(titleFile, System.Text.Encoding.Unicode);

writerTitle.Write(times[titleIndex]);

writerTitle.Write(",");

writerTitle.WriteLine(datas[titleIndex]);

writerTitle.Close();

}

private string getTitleTime(string time)

{

int t = Int32.Parse(time);

string titleOfTime;

switch (t / 6)

{

case 0:

titleOfTime = "midnight";

break;

case 1:

titleOfTime = "morning";

break;

case 2:

titleOfTime = "afternoon";

break;

case 3:

titleOfTime = "evening";

break;

default:

titleOfTime = "errors";

break;

}

return titleOfTime;

}

private bool TotalEffectivenessCheck()

{

for (int i = 0; i < regions.Length; ++i)

{

if ((datas[i] == -1) || (string.IsNullOrEmpty(times[i]))) return false;

}

return true;

}

private bool RegionEffectivenessCheck(string region)

{

if (regions[GetNowIndex()].Equals(region)) return true;

else return false;

}

private bool TimeEffectivenessCheck(string time)

{

if (string.IsNullOrEmpty(time)) return false;

else

{

if (GetNowIndex() == 0) return true;

else if (!string.IsNullOrEmpty(times[GetBeforeIndex()]))

{

if ((Int32.Parse(time) - Int32.Parse(times[GetBeforeIndex()])) < 500) return true;

else return false;

}

else return false;

}

}

private void ResetAll()

{

ResetIndex();

ResetDatas();

ResetTimes();

}

private void ResetDatas()

{

for(int i = 0; i < datas.Length; ++i)

{

datas[i] = -1;

}

}

private void ResetTimes()

{

for (int i = 0; i < times.Length; ++i)

{

times[i] = null;

}

}

private void ResetIndex()

{

index = -1;

}

private int GetNowIndex()

{

return index;

}

private int GetNextIndex()

{

if (index > (regions.Length - 2)) index = -1;

++index;

return index;

}

private int GetBeforeIndex()

{

if (GetNowIndex() == 0) return 0;

else return GetNowIndex() - 1;

}

}

* 매트랩 코드
* 시간별 ping(kr)

clear all; close all;

kr = readmatrix('kr.txt');

krsort = sortrows(kr);

mean = mean(krsort(:,2))

var = var(krsort(:,2))

x=linspace(0,24,6712);

y=krsort(:,2);

semilogy(x,y); grid; xlabel('time (hour)'); ylabel('ping'); title('kr - seoul'); set(gca,'XTick',[0:3:24]); yticks([10 15 20 25 30 40 60 100 150 300 400])

시간별 ping 수치(kr)

clear all; close all;

kr = readmatrix('kr.txt');

krsort = sortrows(kr);

midsum = 0; morsum = 0; aftsum = 0; evesum = 0;

midcnt = 0; morcnt = 0; aftcnt = 0; evecnt = 0;

for i=1:1:6712

if(krsort(i,1)/10000>=0) && (krsort(i,1)/10000<6)

midsum = midsum + krsort(i,2);

midcnt = midcnt+1;

elseif(krsort(i,1)/10000>=6) && (krsort(i,1)/10000<12)

morsum = morsum + krsort(i,2);

morcnt = morcnt+1;

elseif(krsort(i,1)/10000>=12) && (krsort(i,1)/10000<18)

aftsum = aftsum + krsort(i,2);

aftcnt = aftcnt+1;

elseif(krsort(i,1)/10000>=18) && (krsort(i,1)/10000<24)

evesum = evesum + krsort(i,2);

evecnt = evecnt+1;

end

end

midavg = midsum/midcnt

moravg = morsum/morcnt

aftavg = aftsum/aftcnt

eveavg = evesum/evecnt

* 세계지도

lat1 = [-37.8 1.4 35.7 37.7 45.5 38.9 55.8 -26.2 -23.5 52.4];

lon1 = [145 103.9 139.8 127 -73.6 -77 37.6 28 -46.6 4.9];

avg = [167.6 97.7 48.2 15.7 214.8 210.6 198.8 342.7 316.4 255];

Severity = discretize(avg,[0 100 250 400], 'categorical', {'Low', 'Medium', 'High'});

gb = geobubble(lat1,lon1,avg, Severity);

geolimits([-60 80],[30 300])

title 'Ping Rate by Country'

gb.SizeLegendTitle = 'Round Trip Time';

gb.ColorLegendTitle = 'Ping Severity';

* 히스토그램(kr)

clear all; close all;

kr = readmatrix('kr.txt');

krsort = sortrows(kr);

for i=1:1:6712

if(krsort(i,2)>20)

krsort(i,2) = 20;

end

end

y=krsort(:,2);

pd = histogram(y)