Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

ESP32 mikrokontroller hálózati terhelés-vizsgálata

DDoS támadások hatása a szolgáltatásminőségre házi feladat

Készítette

Rádai Ronald

2024

TARTALOMJEGYZÉK

1. Bevezetés

A félév során az általam választott feladat a „DDoS támadások hatása a szolgáltatásminőségre” tárgyból az ESP32 mikrokontroller hálózati terhelésvizsgálata volt. Azért ezt a feladatot választottam, mert szerettem volna jobban megismerkedni az ESP-n futó web szerver képességeivel és korlátaival, hogy későbbi projektjeimben ezt a tudást hasznosítani tudjam. A feladat tervezése során főként azt szerettem volna kideríteni, hogy mekkora (hány csomag/másodperc) terhelést tud kiszolgálni egy ilyen apró, olcsó mikrokontroller.

A megvalósítása során rengeteg új tapasztalatra tettem szert és sikerült megkapnom a kérdéseimre a választ. A megoldást, implementációt élveztem, és a kódokat, illetve az eredményeket egy publikus Github tárolóba töltöttem fel, ezek a következő címről elérhetőek: https://github.com/mineroni/ESP32_DoS

A megvalósítás során az ESP oldalán a webszerver elkészítéséhez az ESP-IDF-et [1] használtam, amely a gyártó által biztosított Framework, kifejezetten nagy hozzáférést biztosít a hardverhez és segítségével alacsony szinten, optimalizáltan lehet kialakítani a kérések kiszolgálásához a szervert.

A PC oldalon .NET-et [2], azon belül pedig C#-ot használtam, mivel a méréseim során nagy számú legitim tartalom hatását szerettem volna vizsgálni a mikrokontroller alapú webszerveremre.

2. Az ESP32 mikrokontroller bemutatása

Az ESP 32 [3] egy kifejezetten olcsó, kínai mikrokontroller, amely rengeteg perifériát kezel és teljesítménye is példás a mikrokontrollerek világában. A működését 2 db Xtensa 32 bites LX6 mikroprocesszor mag biztosítja, amelyek 240 Mhz maximális frekvencián képesek üzemelni.

2.1. Specifikáció

A mikrokontroller specifikációja a következő:

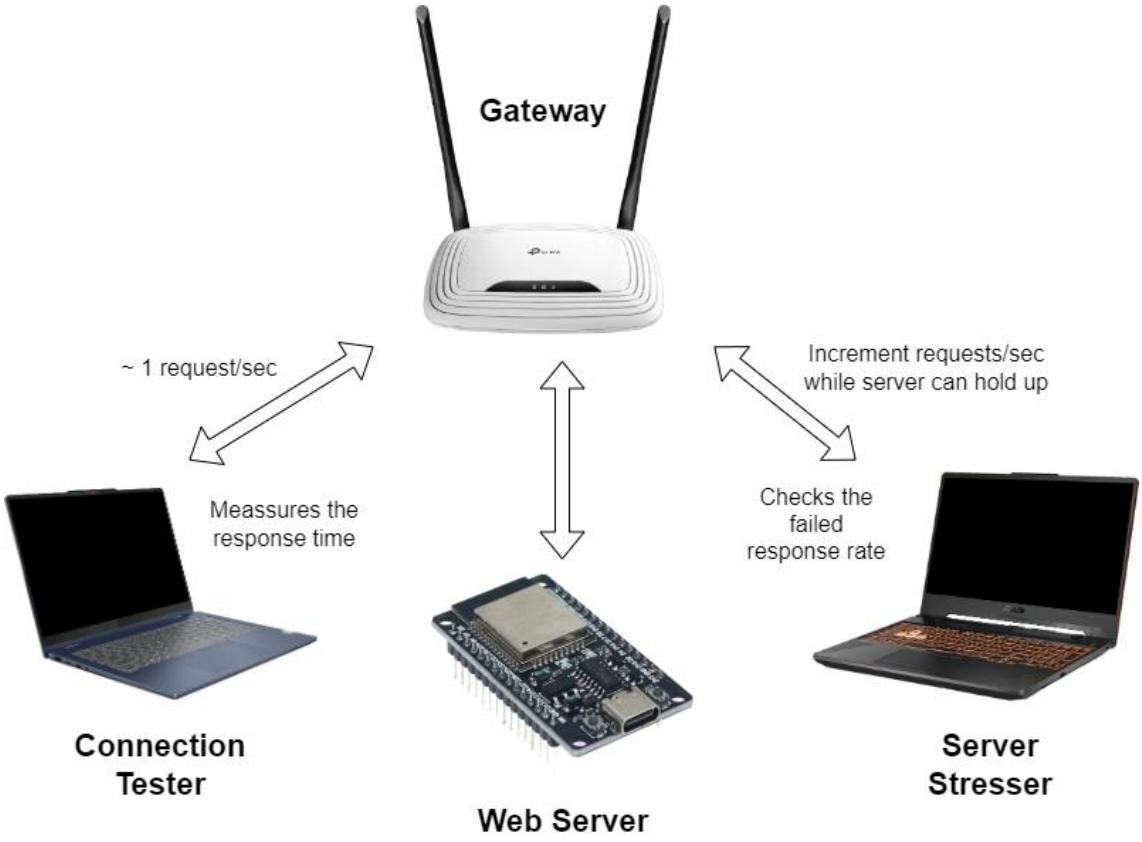
- Xtensa dual-core 32-bites LX6 microprocessor, legfeljebb 240 MHz frekvenciával
- 520 KB SRAM, 448 KB ROM és 16 KB RTC SRAM.
- Támogatja a 802.11 b/g/n Wi-Fi szabványt 150 Mbps sebességig.
- Támogatja a klasszikus Bluetooth v4.2 és a BLE szabványokat.
- 34 programozható GPIO lábbal rendelkezik.
- 18 csatornán 12-bites SAR ADC és 2 csatornán 8-bites DAC együttes kezelése
- Támogatott soros interfacek 4 x SPI, 2 x I2C, 2 x I2S, 3 x UART.
- Motor PWM és legfeljebb 16 csatornás LED PWM.
- Secure Boot és Flash titkosítás.

2.2. Miért az ESP32?

A projekthez azért az ESP32 mikrokontrollert választottam, mert egy kifejezetten széleskörűen használható, olcsó kontroller, amely rengeteg lehetőséget hordoz magában a különböző fejlesztésekhez. Számos eddigi projektemhez használtam már és még továbbra is szeretném majd használni. Mivel a jövőben is tervezek ESP alapon webszervert (kisebb funkciókhoz) megvalósítani, ezért szeretném megtudni, hogy a platform mire képes és meddig lehet kitolni a határait.

3. Megvalósított architektúra

A megvalósítás során az volt az elképzelésem, hogy két különálló PC-ről tesztelem az ESP elérhetőségét és válasz idejét, mivel a két egymástól elszeparált számítógéppel megvalósított architektúra lehetővé teszi, hogy mindenféle torzító hatástól független legyen a rendszer (mint például a gép linksebessége, vagy a csomagok egymáshoz közti sorrendje, eloszlása). A rendszer végleges architektúráját az 1. ábra mutatja be.



1. ábra A mérési környezet architekturális felépítése

Az ábrán látható módon két kliens PC közül az egyik *Stresser* szerepet lát el, vagyis addig növeli a másodpercenkénti kérések számát, amíg a sikertelen kérések aránya drasztikusan meg nem nő. A másik kliens PC *Tester* szerepet lát el, vagyis küld egy kérést a szervernek, miközben méri, hogy mennyi idő alatt ér vissza hozzá a válasz. Minden megérkezett válasz után vár 1 teljes másodpercet, majd küldi a következő kérést. Mindkét számítógépen működik egy logger, amely rögzíti az aktuális mérési adatokat egy időbélyeggel egyetemben, amely segít, hogy a gyűjtött adatok elemzésénél össze lehessen kötni az egyszerre történt eseményeket.

A szerver összeköttetését a kliensekkel egy B/G/N szabványt is támogató router végzi, amely azért fontos, mert az ESP wifi rádiója csak a 2,4 Ghz-es csatornában tud működni.

A Webszerver szerepét egy ESP32 látta el, aminek fel szerettem volna deríteni a mérési kapacitását. A méréseknél szerettem volna szem előtt tartani, hogy a szerver terhelése ne

rosszindulatú túlterhelési szimuláció, hanem legitim forgalom szimulációja legyen. A terhelés szimulációjához a következő végpontokat fejlesztettem, amely segítségével tesztelhető a REST alapú működés:

- Minél rövidebb GET kérés (aminek üres a törzstartalma) és a visszaküldött válasz tartalma is a lehető legrövidebb.
- Olyan GET kérés, aminél egy statikus (135 karakter hosszú) választ ad vissza a szerver.
- Olyan GET kérés, aminél a User-Agent header tartalmát be kell parse-olnia az ESP-nek, és válaszként visszaküldenie egy konstans hello-t, illetve a User-Agent header tartalmát. A tesztelés során a beállított User-Agent header a "C# example" string.
- Olyan GET kérés, amely kiszámoltatja a PI értékét négy tizedesjegyig (nem cache-elve, minden kérésre kiszámítva)
- Olyan POST kérés, ami a body-ban tartalmazott string-et visszaadja a válaszban. Ebben a kérésben a tesztelés során a body tartalma egy 25 hosszúságú, random karakterekből álló string volt.

A megvalósítás során az ESP-t megpróbáltam beállítani úgy, hogy a lehető legtöbb kérést ki tudja szolgálni, tehát nem csinálni felesleges folyamatokat, illetve háttérben futó dolgokat, a webszerverként használt task stackméretét megnövelni, a prioritását maximálisra állítani, illetve maximalizáltam az egyszerre maximálisan nyitva lévő socket-ek számát.

Érdekesség a maximálisan nyitva lévő socket-ek számánál azonban, hogy ez a szám ESP32 mikrokontrollernél (pont úgy, mint az összes többi ESP-nél) maximalizálva van. Az egyszerre nyitott socketek száma az ESP32-nél legfeljebb 10 lehet, amelyből a framework fenntart 3 darabot magának, így 7 darab socket áll rendelkezésre egyidőben a kérések kiszolgálására.

Mérési eredmények elemzésére készítettem egy Python notebook scriptet is, amely matplotlib segítségével kirajzolja a mért adatokat, hogy az könnyen elemezhető legyen.

Minden mérésnél készítettem egy képet Wireshark-ban a szerver IP címére szűrő display filterrel, amely segíthet a különböző következtetések levonásában. Ezen kívül kétféle mérést is végeztem, amelyeket külön-külön plotoltam.

Az egyik mérés a *Tester* node-on futó válaszidő mérése a *Stresser* node működtetése nélkül, a másik pedig a *Tester* és a *Stresser* node mérési eredménye, miközben a *Stresser* node a szerver működési kapacitásának maximumáig terheli a szervert.

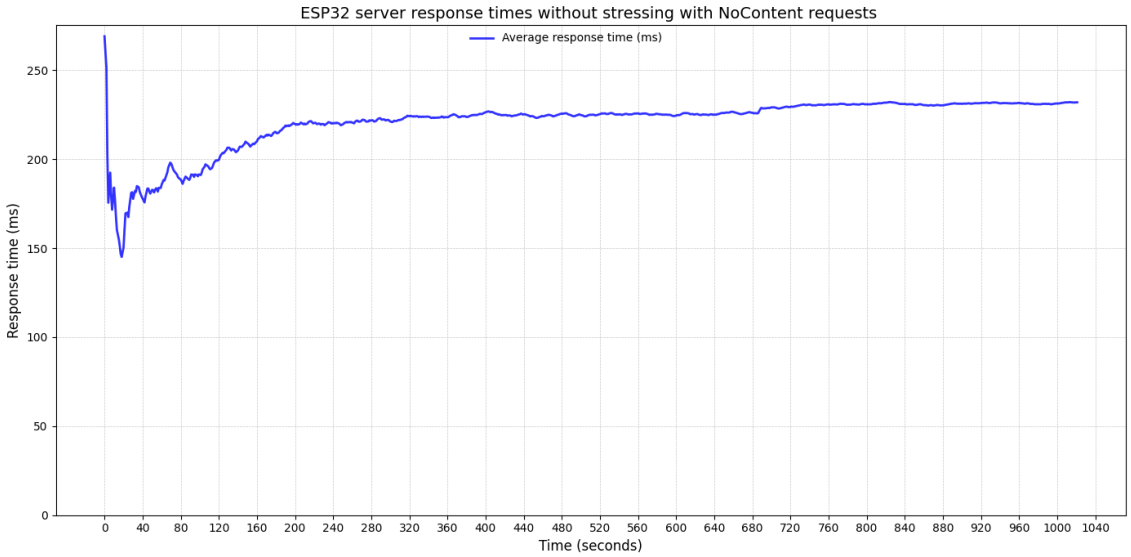
A mérések során a *Stresser* node automatikusan 1 kérés/másodpercről növeli a kérések számát 20 másodpercenként 3 kérés/másodperccel, amíg azt nem érzékeli, hogy a webszerver általi válaszidő, vagy sikertelen válaszok száma drasztikusan megnő. Ekkor visszalép egyet, vagyis 3 kéréssel visszaveszi a kérések számát. Miután ez látszólag stabilizálódott, megnöveli 20 kéréssel a másodpercenkénti kérésszámot, hogy megfigyelje, hogy hogyan reagál a szerver erősen túlterhelt állapotban.

4. Mérési eredmények

A mérések során mind az öt darab korábban említett végpont válaszidejét teszteltem terheléssel és terhelés nélkül is. Ebben a fejezetben a mérések eredményét szeretném bemutatni és értelmezni.

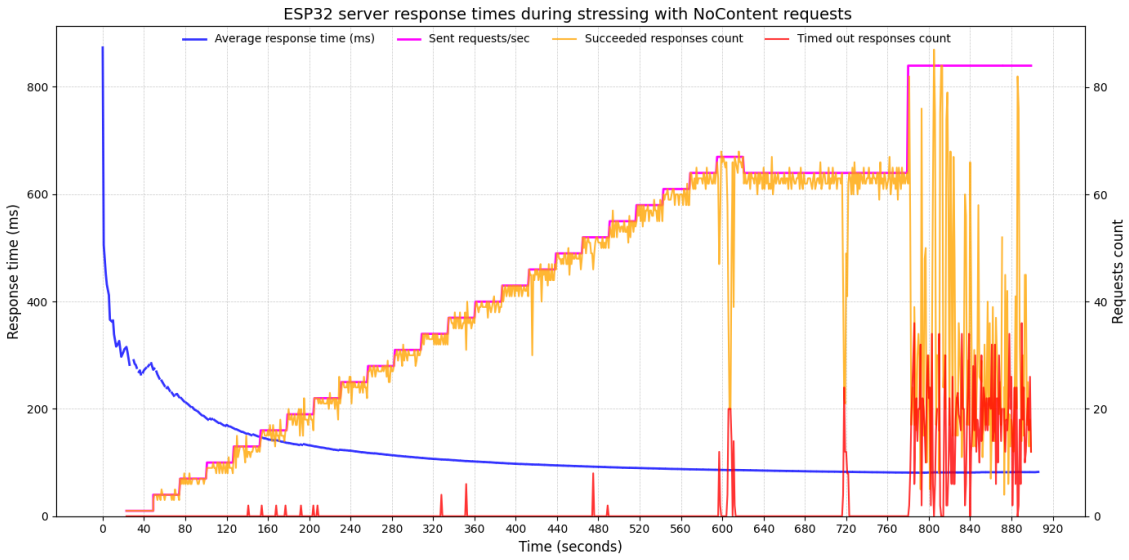
4.1. No Content végpont

A végpont a „/nc” útvonalon érhető el, célja egy minél rövidebb *get* kérést szimulálni (aminek konkrétan üres a törzstartalma és a visszaküldött válasz tartalma is a lehető legrövidebb). A végpont esetében az ilyen kérések hossza 117 byte, a válaszok hossza pedig 56 byte volt. Az átlagos válaszidőt terhelés nélkül a 2. ábra mutatja be.



2. ábra No Content végpont átlagos válaszideje terhelés nélkül

A terheléses mérés során készült log-olt válaszidők és sikeres, hibás kérések eredményét bemutatja a 3. ábra, a TCP által küldött csomagokat, illetve TCP hibák számát pedig a 4. ábra.



3. ábra A No Content végpont átlagos válaszideje, illetve terhelése

4. ábra A terheléses vizsgálat során küldött csomagok, illetve hibák száma a No Content végponttal való kommunikáció során

Az ábrákon látható, hogy a szerver terhelésének eredménye az elvárások szerint alakult, a *Stresser* komponens fokozatosan emeli a terhelést, amíg a hibás válaszok aránya hirtelen ugrásszerűen megnő. Ekkor a kérések számát visszaveszi, aminek hatására a hibás válaszok száma visszaesik. Miután stabilizálódott, akkor egy hirtelen ugrással növeli a kérések számát másodpercenként, amire a szerver kiszámíthatatlanul reagál, a sikeres kérések száma egyik pillanatban nő, a másikban csökken.

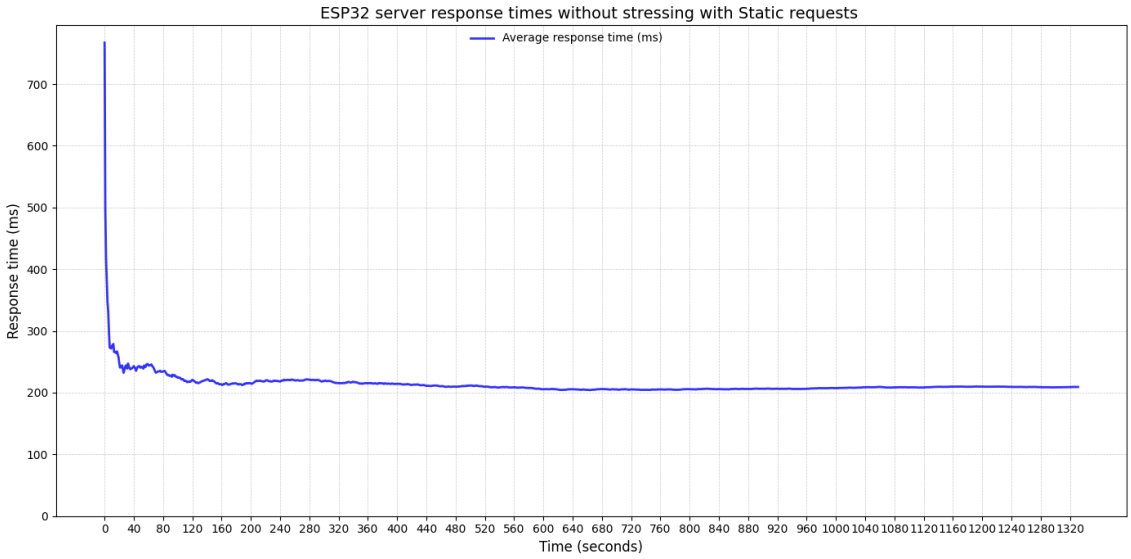
Érdekesség azonban, hogy a *Tester* komponens által tapasztalt válaszidő a növekvő kérések hatására csökken, sőt exponenciálisan csökken. Ennek az okát nem tudtam kétséget kizáróan kideríteni, viszont szerintem nagy valószínűséggel az ESP-n futó FreeRTOS [4]

általi taszkváltások okozzák. Amikor másodpercenként 1-1 kérést intézünk a szerver felé, akkor a taszk nem fog kapni időrést, viszont amikor a szervert megterheljük (nagyságrendileg legalább 10-15 kérés/másodperccel), akkor a taszkváltások száma, így a taszk átlagos válaszideje is sokat csökkenhet. Szerintem ez okozhatja a fentebb említett átlagos válaszidő csökkenést. Fontos megemlíteni, hogy ekkor a válaszidők átlaga valóban csökkent, viszont volt 1-1 kiugró érték, amikor a válaszidő az átlagnak akár a triplája is lehetett.

A stabil másodpercenkénti maximális kérésszám a végpont terhelése közben 65 kérés/másodperc volt, a rendelkezésre álló socketeken. Ekkor az átlagos válaszidő 231 ms-ról nagyságrendileg 82ms-ig csökkenthető.

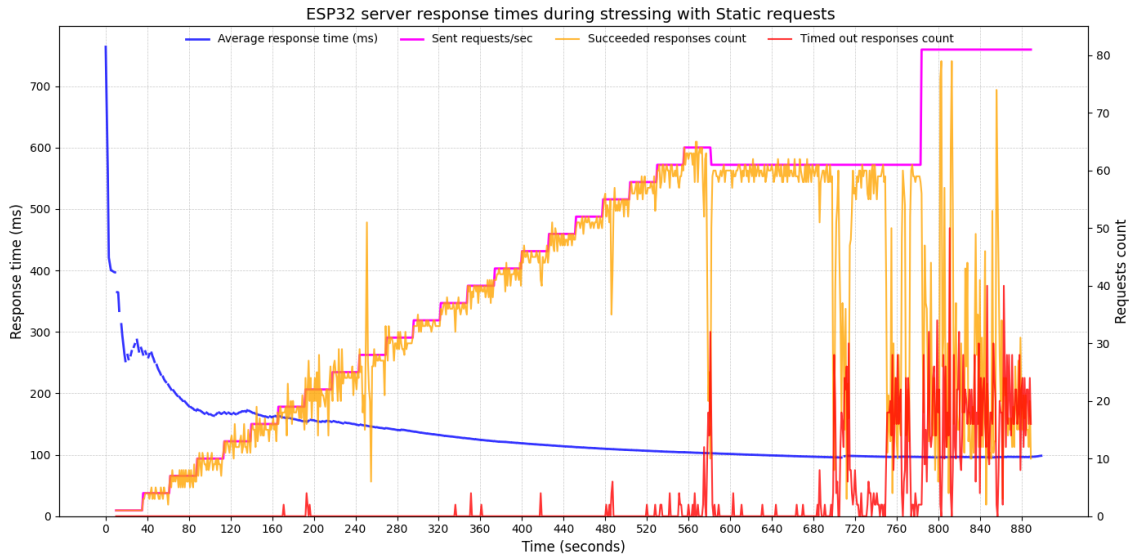
4.2. Statikus kiszolgáló végpont

A végpont a „/” útvonalon, érhető el, célja olyan *get* kérés szimulálása, aminél egy statikus (135 karakter hosszú) választ ad vissza a szerver, ezzel példázva egy rövidebb weboldalt. A végpont esetében az ilyen kérések hossza 115 byte, a válaszok hossza pedig 328 byte volt. Az átlagos válaszidőt terhelés nélkül a 5. ábra mutatja be.

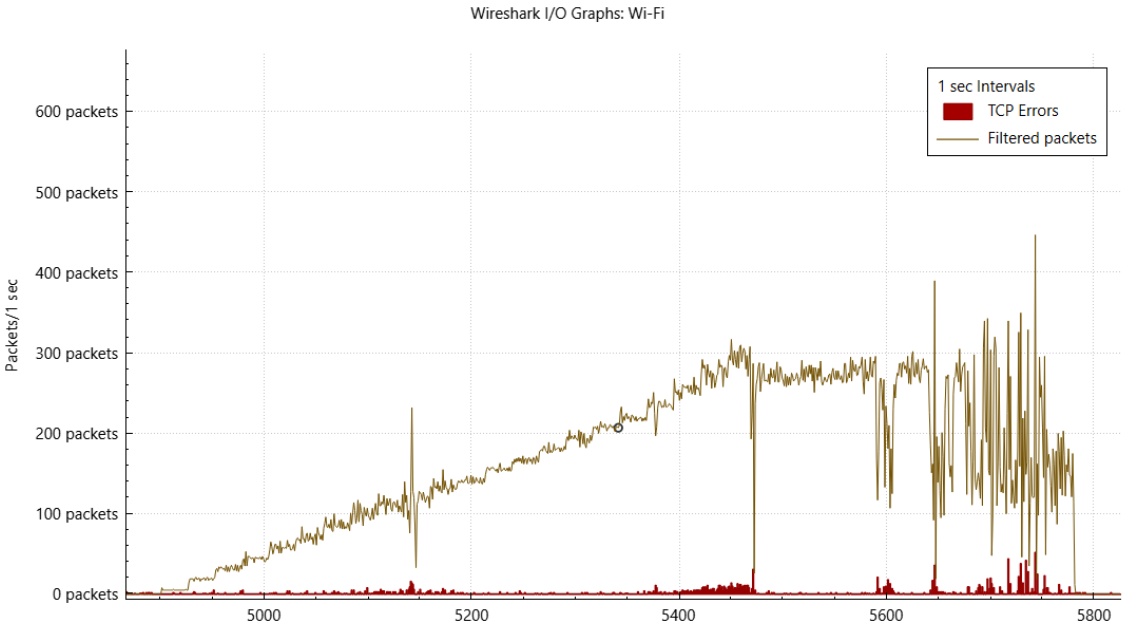


5. ábra A statikus végpont átlagos válaszideje terhelés nélkül

A terheléses mérés során készült log-olt válaszidők és sikeres, hibás kérések eredményét bemutatja a 6. ábra, a TCP szintű sikeres, illetve sikertelen kérések számát pedig a 7. ábra.



6. ábra A Statikus végpont átlagos válaszideje, illetve terhelése



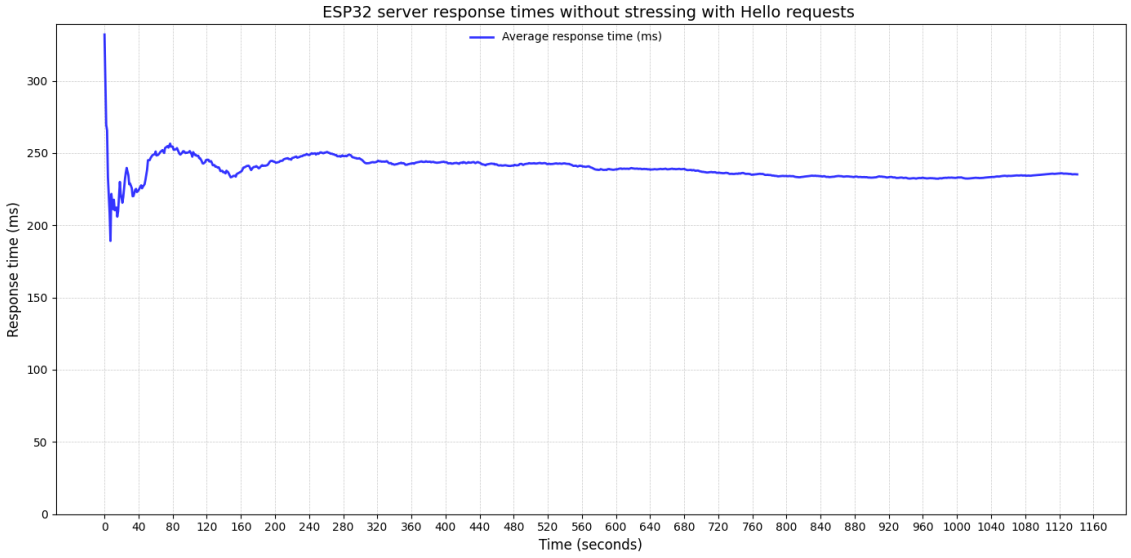
7. ábra A terheléses vizsgálat során küldött csomagok, illetve hibák száma a Statikus végponttal való kommunikáció során

A mérés eredménye ennél a végpontnál nagyon hasonló lett, mint a 4.1. fejezetben tárgyalt No Content végpont esetében. Az érzékelt lassulást nagy valószínűséggel az extra karakterek, amiket vissza kell küldeni és az ezzel járó nagyobb csomaghossz okozta.

A stabil másodpercenkénti maximális kérésszám a végpont terhelése közben 62 kérés/másodperc volt, a rendelkezésre álló socketeken. Ekkor az átlagos válaszidő 209 ms-ról nagyságrendileg 100 ms-ig csökkenthető.

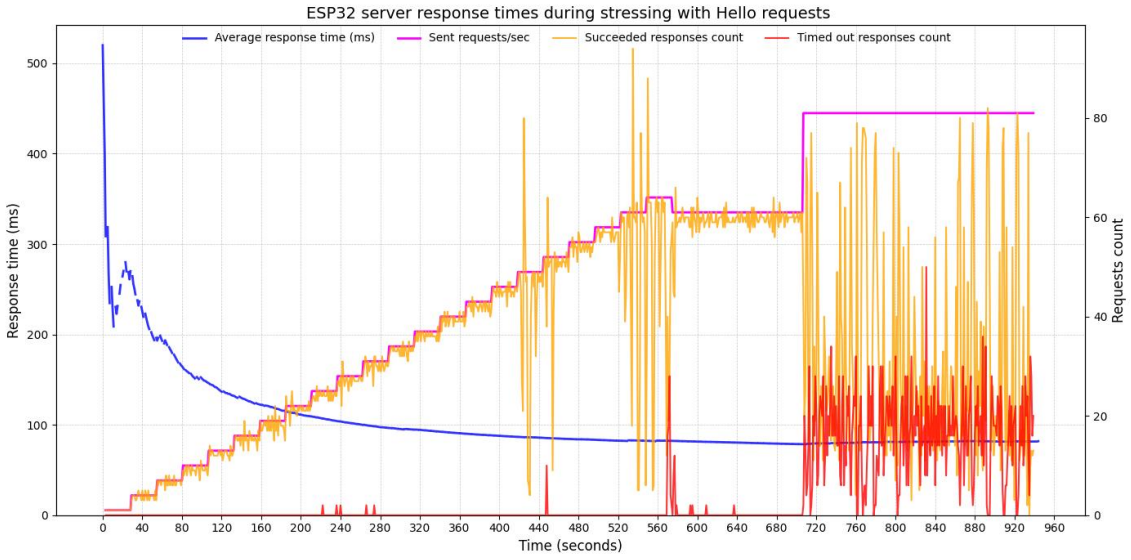
4.3. Hello végpont

A végpont a „/hello” útvonalon érhető el, célja olyan *get* kérés, aminél a User-Agent header tartalmát be kell parse-olnia az ESP-nek, és válaszként visszaküldenie egy konstans "Hello" stringet, illetve a User-Agent header tartalmát. A tesztelés során a beállított User-Agent header a "C# example" string volt. A végpont esetében az ilyen kérések hossza 120 byte, a válaszok hossza pedig 73 byte volt. Az átlagos válaszidőt terhelés nélkül a 8. ábra mutatja be.

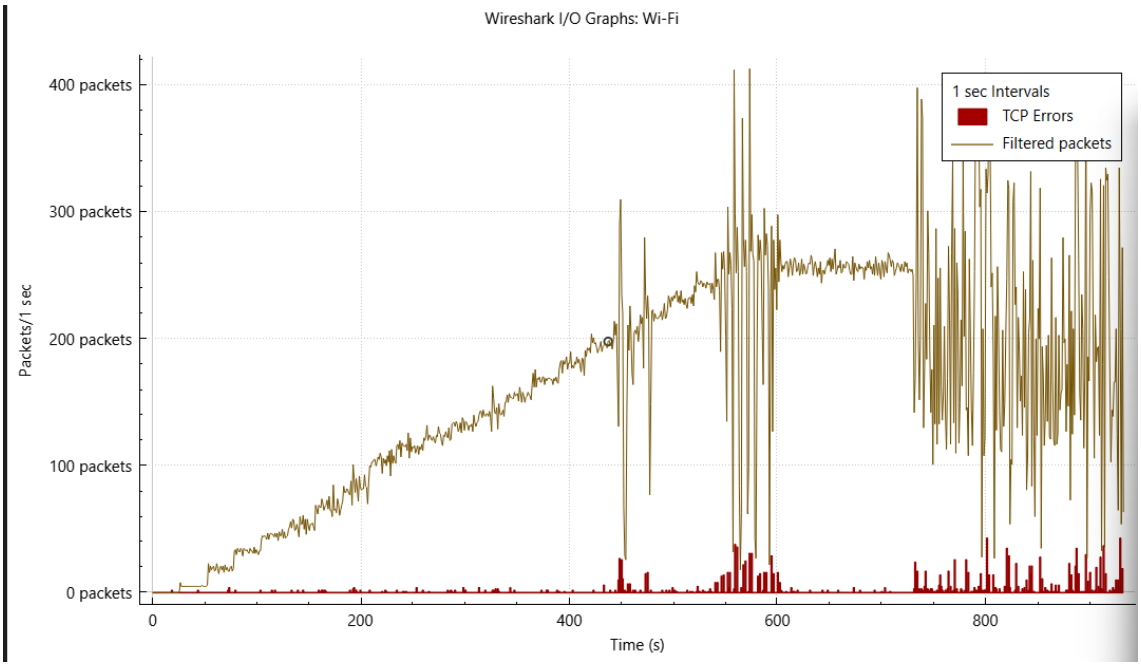


8. ábra A Hello végpont átlagos válaszideje terhelés nélkül

A terheléses mérés során készült log-olt válaszidők és sikeres, hibás kérések eredményét bemutatja a 9. ábra, a TCP szintű sikeres, illetve sikertelen kérések számát pedig a 10. ábra.



9. ábra A Hello végpont átlagos válaszideje, illetve terhelése



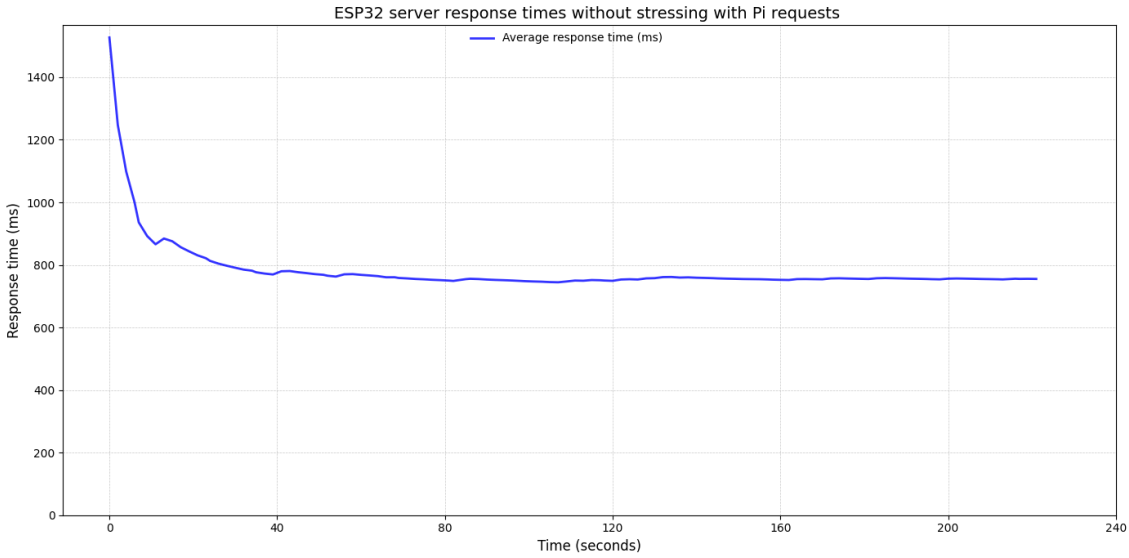
10. ábra A terheléses vizsgálat során küldött csomagok, illetve hibák száma a Hello végponttal való kommunikáció során

A mérési eredmények nagyon hasonlítanak az eddigi eredményekhez, érdemes megjegyezni, hogy a processzor számára nem igazán jelentett problémát az extra feldolgozás, vagyis a kérésben szereplő extra header parsolása és a válaszba fűzése. Ennek az oka az lehet, hogy a szűk keresztmetszetet az eddigi végpontoknál valószínűleg a nagyban korlátozó maximális 7 db nyitott socket száma okozta, nem pedig a processzor terheltsége, vagy a hálózati leterheltség. Ebben az esetben pedig nagy valószínűséggel rendelkezésre áll az extra erőforrás a parsolásra.

A stabil másodpercenkénti maximális kérésszám a végpont terhelése közben 62 kérés/másodperc volt, a rendelkezésre álló socketeken. Ekkor az átlagos válaszidő 236 ms-ról nagyságrendileg 82 ms-ig csökkenthető.

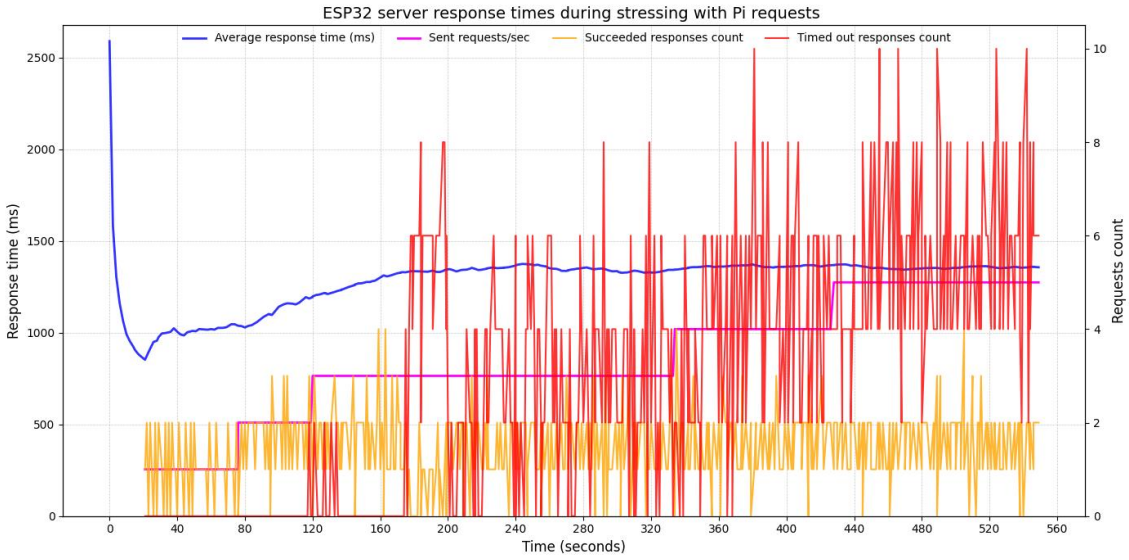
4.4. Pi végpont

A végpont a „/pi” útvonalon érhető el, célja egy olyan *get* kérés implementálása, amely kiszámoltatja a PI értékét négy tizedesjegyig (nem cache-elve, minden kérésre kiszámítva), ezzel az eddigi végpontoknál jóval nagyobb processzorterhelést szimulálva. A végpont esetében az ilyen kérések hossza 117 byte, a válaszok hossza pedig 105 byte volt. Az átlagos válaszidőt terhelés nélkül a 11. ábra mutatja be.

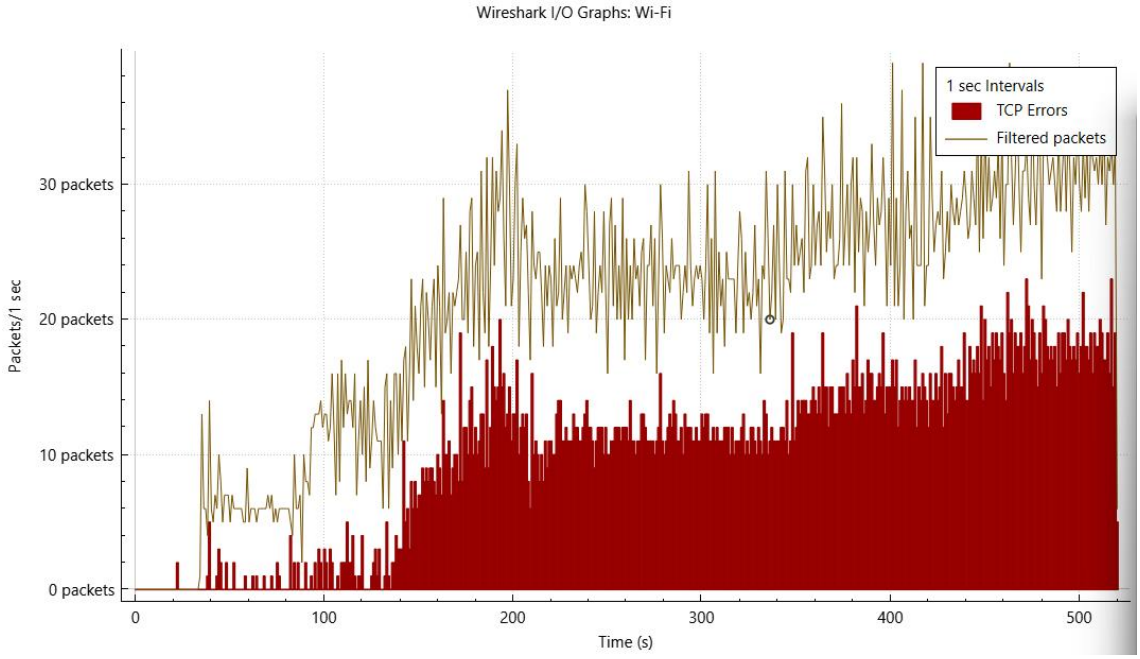


11. ábra A Pi végpont átlagos válaszideje terhelés nélkül

A terheléses mérés során készült log-olt válaszidők és sikeres, hibás kérések eredményét bemutatja a 12. ábra, a TCP szintű sikeres, illetve sikertelen kérések számát pedig a 13. ábra.



12. ábra A Pi végpont átlagos válaszideje, illetve terhelése



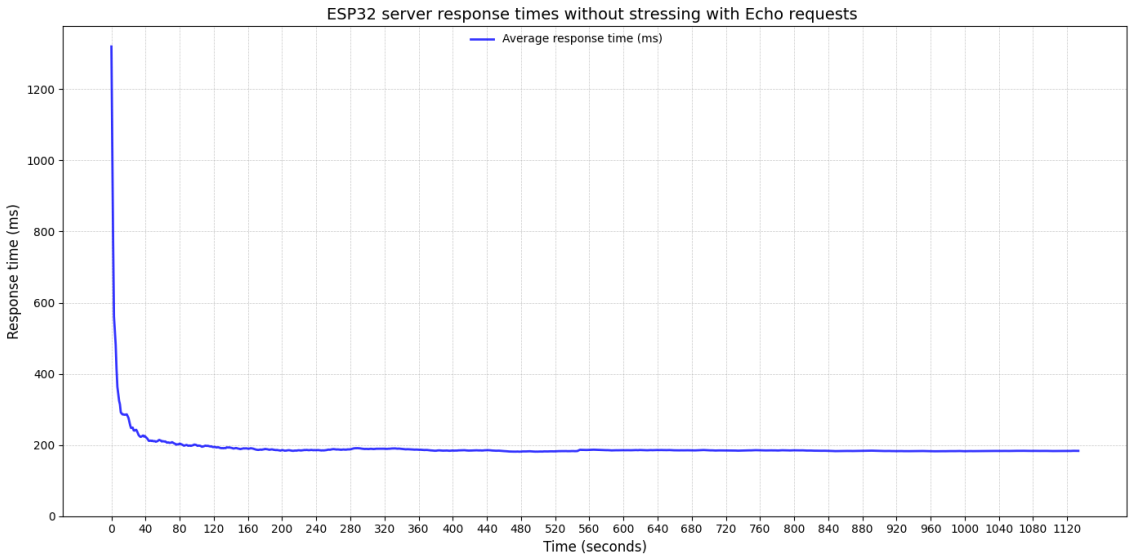
13. ábra A terheléses vizsgálat során küldött csomagok, illetve hibák száma a Pi végponttal való kommunikáció során

Ezen végpont mérésének eredménye kifejezetten érdekesre sikeredett. Ennek az oka az lehet, hogy itt a teljes mérés időtartama alatt nem sikerül a maximálisan felhasználható 7 nyitott socket fölé vinni a terhelést, mivel a végpontokon a számítási feladatok annyira hosszúak (sok ideig tart) voltak. Ez azt okozta, hogy a szűk keresztmetszet a processzor erőforrása lett. Ez megfigyelhető többek között a kiemelkedően magas hibaarányon, illetve az átlagos válaszidő növekedésén is. Érdekesség továbbá, hogy miután a kérések száma maximalizálódott (2-3 plusz a *Tester* kérései) az átlagos válaszidő megközelítően nem emelkedett. Ennek az oka az, hogy a processzor csak az általa kiszolgálható kéréseket látta el, a többit eldobta.

A stabil másodpercenkénti maximális kérésszám a végpont terhelése közben 2-3 kérés/másodperc volt. Ekkor az átlagos válaszidő a terhelés nélküli 0,8 másodpercről 1,2 másodpercre nőtt a terhelés hatására.

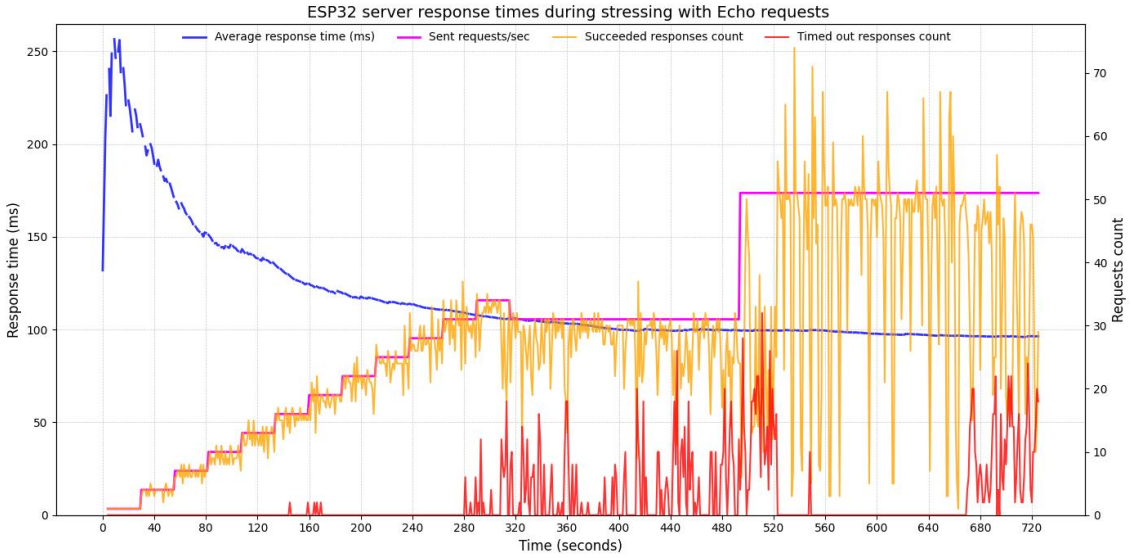
4.5. Echo végpont

A végpont a „/echo” útvonalon érhető el, célja egy olyan *post* kérés implementálása, ami a body-ban tartalmazott string-et visszaadja a válaszban. Ebben a kérésben a tesztelés során a body tartalma egy 25 hosszúságú, random karakterekből álló string volt. A végpontnak a segítségével ellenőrizni tudjuk, hogy a kontroller hogyan birkózik meg a *post* típusú kérésekkel. A végpont esetében az ilyen kérések hossza 206 byte, a válaszok hossza pedig 81 byte volt. Az átlagos válaszidőt terhelés nélkül a 14. ábra mutatja be.

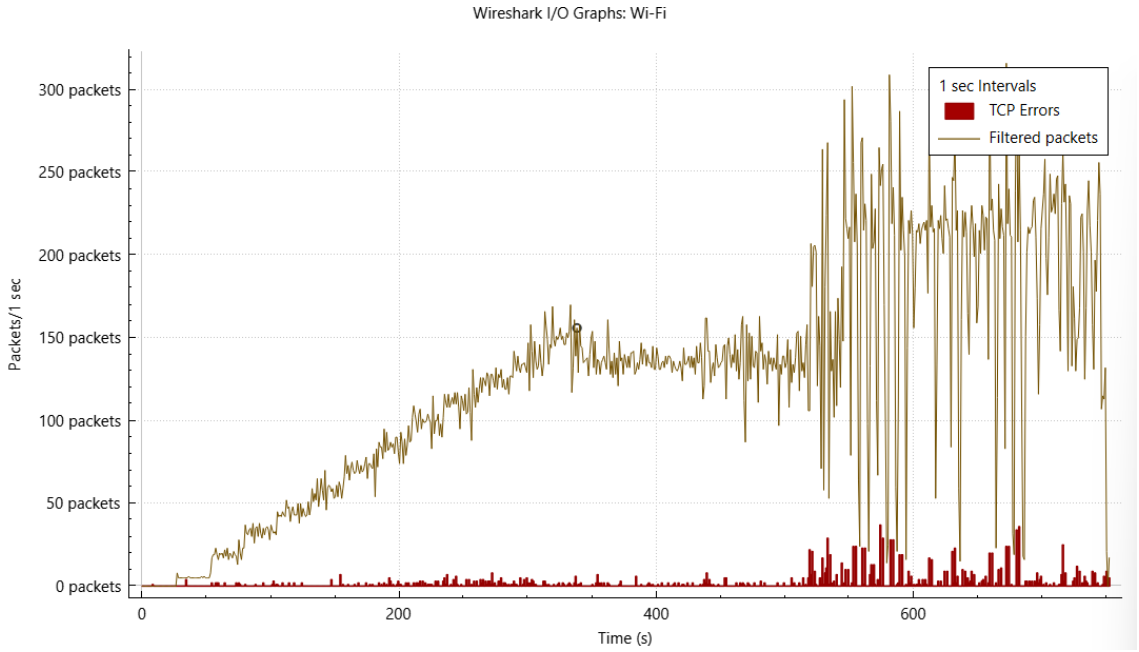


14. ábra Az Echo végpont átlagos válaszideje terhelés nélkül

A terheléses mérés során készült log-olt válaszidők és sikeres, hibás kérések eredményét bemutatja a 15. ábra, a TCP szintű sikeres, illetve sikertelen kérések számát pedig a 16. ábra.



15. ábra Az Echo végpont átlagos válaszideje, illetve terhelése



16. ábra A terheléses vizsgálat során küldött csomagok, illetve hibák száma az Echo végponttal való kommunikáció során

A *post* kérések mérése ismét érdekes eredményt hozott. A szerver által maximálisan kiszolgált kérdések száma pont a fele a *get* kérések által maximálisan kiszolgálhatónak. Ennek a pontos okát sajnos nem sikerült megtalálnom, de nagy valószínűséggel az ok a kérések kezelésében, annak implementációjában keresendő.

A stabil másodpercenkénti maximális kérésszám a végpont terhelése közben 31 kérés/másodperc volt, a rendelkezésre álló socketeken. Ekkor az átlagos válaszidő 181 ms-ról nagyságrendileg 100 ms-ig csökkenthető.

5. Összefoglalás

A készített és futtatott tesztek néhol meglepő, ám többnyire megmagyarázható eredményekkel zárultak. Néhány érdekes részletre is sikerült fényt deríteni a munkám során, amely a jövőben segíteni fog jól tervezni, vagyis, hogy milyen tervezett terhelés esetén érdemes és milyen terhelés felett nem érdemes egy ESP32 mikrokontrollert használni, mint webszerver.

A munkám során elért eredményeim röviden összefoglalva:

- Az ESP32 mikrokontrolleren amennyiben nem kiemlekedően számításigényesek a végpontok folyamatai, akkor intuitív várakozásainkkal ellentétben a szűk keresztmetszetet nem hálózati elérés, nem a processzor, hanem az egyidejűleg maximálisan nyitott socketszám (melynek a maximális értéke 7 db) fogja jelenteni.
- A *post* kérések kétszer olyan erőforrásigényesek az ESP számára, mint a *get* kérések.
- A maximálisan egyidejűleg az ESP32 által fogadott csomagszám nagyságrendileg 300 csomag/másodperc *get* kérések esetén, illetve 150 csomag/másodperc *post* kérések esetén.
- Ha a mikrokontroller processzora nincsen kihasználva, akkor a legitim válaszokra jutó átlagos válaszidő csökkenthető, ha beinjektálunk néhány kérést véletlenszerűen.

Ezeken kívül fontos megjegyeznem, hogy a mérések eredménye a meglévő kliensekre igaz. A *Stresser* komponens fejlesztése során észrevettem, hogy amikor a webszerver erőteljesen le van terhelve, akkor egy új, másik kliensnek nem fog válaszolni, mivel mind a 7 webszerver célra felhasználható socketje foglalt lesz, így a hozzá intézett kéréseket ilyen helyzetben egyszerűen eldobja a processzor.

6. Hivatkozások

[1] E. Systems, „Welcome to Espressif IoT Development Framework!,” Espressif Systems, [Online]. Available: https://idf.espressif.com.

[2] Microsoft, „Build it with .NET,” Microsoft Corporation, [Online]. Available: https://dotnet.microsoft.com/en-us/.

[3] E. Systems, „ESP32,” [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf.

[4] FreeRTOS, „FreeRTOS™,” Amazon Web Services, [Online]. Available: https://www.freertos.org.