DCM2 LAB: Performance van ESP32 TCP/IP stack

Auteur: Victor Hogeweij

Docenten: Ruud Elsinghorst
Remko Welling

Klas: ESE-2A

Instituut: Hogeschool Arnhem-Nijmegen

Chapter 1

Versiegeschiedenis

Versie	Datum	Persoon	Notitie/verandering
1	20-11-22	VH	Opzet verslag
2	24-11-22	VH	Toevoegen hoofdstukken
3	26-11-22	VH	Introductie schrijven
4	26-11-22	VH	Schrijven van Introductie en opzet achtergrond
5	27-11-22	VH	Schrijven van sectie TCP/IP stacks
6	1-12-22	VH	Schrijven van H3.1 en H3.2
7	1-12-22	VH	Schrijven van H4.1 en H4.2
8	3-12-22	VH	Schrijven van H4.2.2
9	4-12-22	VH	Schrijven van H4.2.3, H4.2.4 en H4.2.5
10	4-12-22	VH	Toevoegen illustrerende figuren bij tekst in H3
11	7-12-22	VH	Schrijven van H5
12	11-12-22	VH	Toevoegen testresultaten in H5

Contents

Chapter 2

Introductie

Wereldwijd komen er elk jaar steeds meer embedded systemen bij met internet functionaliteit. Dit brengt naast een hoop mooie mogelijkheden voor de industrie, ook een hoop uitdagingen mee voor fabricanten en ingenieurs. De grootste uitdaging van deze embedded systemen is om een stabiele softwarebasis te schrijven die de hardware assisteert bij het maken en in stand houden van de verbinding. De basis van deze software is een tcp/ip stack. De standaarden en technische eisen van de stack staan vastgelegd, de implementatie echter verschilt. Om de prestaties van een embedded systeem met netwerk functionaliteit vastteleggen zijn uitgebreide testen nodig.

2.1 Doel en motivatie

Het doel van dit onderzoek is het uitzoeken welke prestaties behaald kunnen worden op een veel voorkomend embedded systeem. Dit onderzoek zal met behulp van Iperf performance testen vastleggen wat de prestaties zijn met verschillende verbindingsparameters.

De resultaten zullen helpen bij het vastleggen van de relatie tussen de verbindingsparameters en bandbreedte.
Dit bescheven werk zal de nadruk leggen op de werking en het testen van tcp/ip stacks op embedded systemen. De testprocedure voor dit onderzoek kan ook nageproduceerd worden op een generieke computer.
De motivatie voor dit werk is de eigen interesse voor computernetwerken en embedded systemen.

2.2 Onderzoeksvragen

Dit onderzoek zal zich richten op het uitzoeken welke prestaties behaald kunnen worden op een ESP32 microcontroller van het merk Espressif. In het onderzoek zullen de hoofvraag en deelvragen beantwoord worden.

2.2.1 Hoofdvraag

De hoofdvraag voor dit onderzoek is:
"What performance can be achieved with the TCP/IP stack on the ESP32?"

2.2.2 Deelvragen

De deelvragen aanvullend op de hoofdvraag zijn:
"What is the most efficient package size?"
"What is the maximum bit rate?"
"What applications can be supported with this system?"

Chapter 3

Achtergrond

Dit hoofdstuk gaat kort in op achtergrond informatie die benodigd is om bepaalde delen van het onderzoek te kunnen begrijpen.
De eerste sectie van dit hoofdstuk geeft de benodigde achtergrond informatie over embedded systemen.
De tweede sectie gaat over tcp/ip stacks
De derde sectie gaat over sockets

3.1 Embedded systemen

Embedded systemen is een woord van de laatste jaren David Stepner and Hui, 1999, maar embedded systemen bestaan al veel langer. Het enige wat nodig is, is een blik werpen op de apparaten om je heen: Telefoons, Modems, Televisies en koffie apparaten om een paar voorbeelden te noemen.
Deze paragraaf geeft een korte introductie in embedded systemen.

3.1.1 Het verschil tussen een computer en embedded systeem

Een embedded systeem wordt vaak beschreven als een systeem die een bepaald aantal vaste taken moet uitvoeren met beperkte ingebouwde functionaliteit (David Stepner & Hui, 1999). Dit zijn bijna altijd onzichtbare mini computers die ingebouwd zitten in apparaten.
Maar het kunnen ook chips met programmeerbare hardware zijn (FPGA's of CPLD's).
Het grote verschil tussen een computer en een embedded systeem is het doeleind. Embedded systemen zijn ontworpen voor een specifieke taak en zijn vaak alleen goed in deze taak. Normale computers daarentegen zijn gemaakt om verschillende uiteenlopende taken goed uittevoeren. Doordat een embedded systeem vaak maar een taak heeft zijn embedded systemen vaak uitgerust met veel minder geheugen en (algemene) computerkracht dan een normaal computersysteem.

3.1.2 Het ontwerp van embedded systemen

Zoals hierboven toegelicht zijn embedded systemen vaak ontworpen voor een specifiek aantal vaste taken. Zaken waar vaak rekening mee gehouden moet worden bij het ontwerpen van een embedded systeem zijn onder andere: reactie tijd nauwkeurigheid, formaat, energie verbruik en kosten (Jacob, 2000).

Reactie tijd is een belangrijk begrip in het embedded domein. Of het hier gaat om een taak die op een bepaalde tijd wordt uitgevoerd (zoals een alarmklok) of de tijd tussen twee taken (zoals bij het geven van medicatie via infuuspompen) het kan een heel grote rol spelen in het ontwerp van het embedded systeem. Het moeilijkst is dan ook om deze tijden te bepalen, en vervolgens te bepalen of het strakke deadlines zijn die niet overschreden mogen worden. Zodat vervolgens in het ontwerp zoveel mogelijk gedaan kan worden om deze tijden te waarborgen.

Formaat van het systeem is ook een belangrijke weging. Veel embedded systemen die in dezer dagen verkocht worden, worden vaak verkocht omdat ze kleiner zijn dan hun voorganger(s). Neem als voorbeeld de smartphone, deze is door de jaren heen steeds kleiner en populairder geworden.

Een andere belangrijke weging in het ontwerp van een embedded systeem is energieverbruik. Verder gaande op wat hier boven staat, worden veel embedded systemen kleiner. De groei van energiedensiteit van accu's en batterijen kunnen de krimp (in formaat) niet bijbenen. Met als gevolg dat embedded systemen zuiniger moeten worden om op een kleinere batterij of accu te kunnen werken.

Ten slotte het laatste punt kosten. Ondanks alle punten hierboven, als het systeem heel veel kost zal het niet verkopen. De meeste eindgebruikers leveren graag prestaties of batterijduur in om een goedkoper product te hebben. Bij de ontwerpfase is het zeer belangrijk om het kostenplaatje in de gaten te houden bij een toevoeging aan of modificatie van het ontwerp.

3.2 TCP/IP stacks

Het tcp/ip model is een van de fundamentele bouwstenen van het huidige internet infrastructuur. Het model bestaat uit vier lagen, zie figuur 1.
De functies van de tcp/ip lagen zijn: - De netwerk laag: Deze laag komt overeen met de fysieke en data link laag in het osi model. Deze laag is verantwoordelijk voor de data transmissie over een netwerk, het definieert hoe twee apparaten fysiek met elkaar moeten communiceren.
- De internet laag: Deze laag stuurt data van het bron apparaat naar apparaten in het pad tussen het bron en doel apparaat. Het IP protocol neemt het grootste deel van deze taak op zich.
- De transport laag: Deze laag zorgt ervoort dat de data bij de juiste applicatie geleverd wordt. Er zijn twee protocollen die deze taak kunnen vervullen: TCP

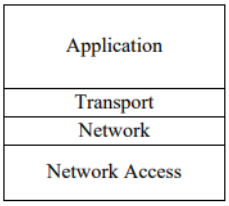


Figure 3.1: TCP/IP model

en UDP. TCP is verantwoordelijk voor het bieden van een betrouwbare, verliesloos connetie georienteerde verbinding. UDP daarentegen is verantwoordelijk voor onbetrouwbare connectieloze verbinding tussen twee hosts.
- De applicatie laag: Deze laag verzorgt de communicatie tussen de transport laag en de software applicaties. Dit doet de laag met protocollen zoals: HTTP, HTTPS, FTP, SMTP, enz.

3.2.1 Implementatie van TCP/IP stacks

Er zijn twee soorten TCP/IP stack implementaties; Een is de TCP/IP stack afgeleid van de BSD implementatie (M. K. McKusick & Quarterman, 1996) en de andere is een door de embedded systeem fabrikant zelf geimplementeerde versie (Dunkels, 2003).
BSD is een besturingssysteem afgeleid van het UNIX besturingssysteem. BSD wordt nog weleens gebruikt op workstations en servers. Maar het was vroeger prominent aanwezig, doordat het zo prominent aanwezig was, hebben veel besturingssystemen zoals Linux met een kleine aanpassing de BSD implementatie voor de TCP/IP stack geadopteerd.
Embedded systemen kunnen vaak door hardware limitaties niet de volledige BSD TCP/IP stack gebruiken. Om het toch werkend te krijgen op de hardware wordt de TCP/IP stack door de fabrikant van het embedded systeem versimpeld (Wang, 2021). Ondanks dat de TCP/IP stack versimpeld is, probeert de fabrikant toch de TCP/IP stack zoveel mogelijk compatibel te maken met de volledige TCP/IP stack die te vinden is in normale besturingssystemen.
De manier waarop de TCP/IP stack versimpeld kan worden, is het optimaliseren naar de hardware limitaties van het embedded systeem.

Neem als voorbeeld een webserver:

Een webserver is gebouwd met een simpele webinterface met alleen een paar knoppen erop. Deze implementatie heeft alleen de protcollen: HTTP, TCP, RARP, ARP en ICMP nodig. Protocollen zoals FTP en SNMP zijn in deze implementatie niet nodig. Deze protocollen kunnen dan ook verwijderd worden uit de broncode om betere prestaties en een kleinere geheugen footprint te creeren (Wang, 2021).

Een ander voorbeeld van een versimpelde tcp/ip stack is lwIP (Dunkels, 2001). lwIP is zodanig gemodificieerde tcp/ip stack dat het zelfs op 8- of 16-bit microcontrollers kan draaien. De manier waarop dit bereikt wordt is door bepaalde functionaliteit weg te laten. Zo heeft lwIP alleen maar ondersteuning in het IP protocol voor versturen, ontvangen en doorsturen van paketten. Het ondersteund niet het verwerken en reconstrueren van gefragmenteerde IP paketten. Voor het tcp protocol zijn er ook dingen weggehaald of versimpeld om het minder intensief te maken voor het embedded systeem waar het op draait.

3.3 Sockets

Sockets zijn vaak onderdeel van een TCP/IP stack API die het mogelijk maakt om met behulp van de TCP/IP stack een directe TCP of UDP verbinding te maken met een andere host. Ook socket API's zijn vaak afgeleid van de BSD implementatie, maar het kan ook een custom implementatie zijn gemaakt door de fabrikant van het embedded systeem.

Een socket bestaat uit een IP-adres en een poort nummer. Sockets hebben geen bron adres nodig, standaard zijn sockets geconfigureerd om alleen te sturen en niet te ontvangen. Om toch data te ontvangen, kan een socket zichzelf vastmaken(binden) aan een lokaal adres.

Chapter 4

Onderzoeksmethode

Dit hoofdstuk licht de onderzoeksmethode en testopstelling toe.

4.1 ESP32

Het platform dat onderzocht wordt is de ESP32. De ESP32 is een microcontroller van de Chinese fabrikant Espressif.
De microcontroller bevat een WiFi en Bluetooth (met ble) peripheral die met behulp van de open source sdk aangestuurd kan worden. De ESP32 is een veelgebruikte microcontroller in de hobbyscene. Dit komt voornamelijk door de goede ondersteuning voor de arduino sdk en de ingebouwde WiFi peripheral.

Andere redenenen voor de populariteit van deze microcontroller zijn (Espressif, n.d.-a):
- De twee xTensa lx6 kernen
- 448 Kb ROM (uitbreidbaar tot 16MB met een SPI flash chip)
- 520Kb SRAM
- Freertos ondersteuning
- Cryptographische functies voor hardware acceleratie van de tcp/ip stack en beveiligingsalgorithmen
- Zowel in module vorm als losse chip leverbaar.

Dit onderzoek zal gebruik maken van een ESP32 development board, het ESP32 development board heeft alle randcomponenten benodigd om de chip te programmeren en voeden.

4.1.1 TCP/IP stack en Programmeer platformen

Het platform ESP32 heeft verschillende soorten tcp/ip stacks en programmeer platformen. Zo zijn er programmeer platformen als: Mongoose os, Zerynth,

NodeMCU en ESP-IDF met freertos.
Het platform wat gekozen is voor dit onderzoek is ESP-IDF met freertos.
Bij dit platform zit een tcp/ip stack meegeleverd, de tcp/ip stack die is meegeleverd is een aangepaste versie van de lwIP(lightweight tcp/ip stack) tcp/ip stack (Espressif, n.d.-c).
De aanpassingen zijn gedaan door de fabrikant zijn om lwIP beter te integreren in de bestaande software van de ESP-IDF SDK.
Voorbeelden van modificaties die zijn aangebracht aan de lwip library zijn onder andere:

- Het thread safe maken van sockets
- Timers uitzetten als er geen gebruik van wordt gemaaakt of als er een timeout situatie optreed
- IPV4 en IPV6 multicast socket opties toevoegen
- Het toevoegen van aan IPV4 gekoppelde IPV6 adressen
- Het toevoegen van geadvanceerde configureerbare hooks aan het build systeem.

Naast dat de ESP32 een tcp/ip stack gebruikt, gebruikt de andere host waarmee de ESP32 verbindt ook een TCP/IP stack. Bij dit onderzoek is de andere host een PC met een Manjaro Linux variant. Deze host maakt gebruik van de Linux Kernel TCP/IP stack. Deze TCP/IP stack is een direct afgeleide versie van de BSD TCP/IP stack en ondersteunt ook de volledige BSD TCP/IP stack. Ook in het linux ecosysteem zijn er andere opties voor TCP/IP stack, zoals PF-RING, Snabbswitch, DPDK en Netmap (Majkovski, n.d.). Om praktische redenen wordt er in dit onderzoek gebruik gemaakt van de standaard met linux meegeleverde Linux kernel tcp/ip stack.
De compiler die gebruikt wordt in dit onderzoek voor de ESP32 is de Xtensa ESP32 GCC compiler (2021r2-8.4.0). In het build systeem is er gebruik gemaakt van CMake(3.20.3) en Ninja(1.10.2).
De compiler die gebruikt wordt in dit onderzoek op de PC is GCC(12.2.0). In het build systeem is er gebruikt gemaakt van cmake(3.24.3) en ninja(1.11.1). Voor scripting wordt er gebruik gemaakt van Python(3.10.8).

4.2 Testmethode

4.2.1 iperf

Om op een industriestandaard wijze te kunnen testen wordt er gebruik gemaakt van het programma Iperf om de bandbreedte testen uit te kunnen voeren.
Deze manier van testen kan het hele systeem (WiFi peripheral, memory management en tcp/ip stack) testen. Bovendien heeft iperf opties om package sizes, window sizes en andere variabelen gebruikt in dit onderzoek aan te kunnen passen. Om Iperf te kunnen gebruiken op de ESP32 wordt er gebruik gemaakt van het Iperf voorbeeld geleverd door de fabrikant (**espiperfexample**). Dit voorbeeld levert een vrijwel complete implementatie van Iperf v2.

Iperf werkt door een verbinding tussen twee hosts te maken met tcp of udp sockets en vervolgens pakketten met een vaste grootte over deze verbinding te sturen gedurende een bepaalde tijd. De tijd is instelbaar en is normaal 10 seconden. In deze 10 seconden houdt het programma bij hoeveel pakketten er verstuurd zijn (en goed ontvangen) en bepaald hieruit wat de bandbreedte is van de verbinding.

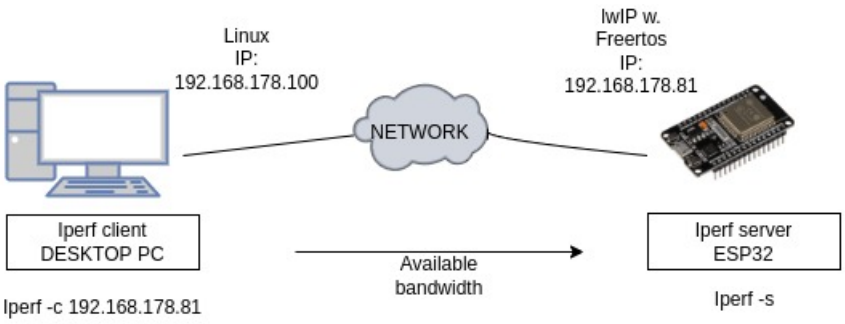


Figure 4.1: Iperfs werking met ESP32 als server

4.2.2 Negatieve factoren

Er zijn verschillende factoren die de performance van de tcp/ip stack negatief kunnen beïnvloeden. Voorbeelden van deze factoren zijn ruis over het medium, de snelheid van de interne communicatiebus tussen de PHY en de microcontroller, de snelheid van het geheugen en het antenne ontwerp van de WiFi controller.

Espressif, de fabrikant van de ESP32 heeft dezelfde testen uitgevoerd (als dit onderzoek) met behulp van Iperf. Waarbij de testen in zowel een open ruimte

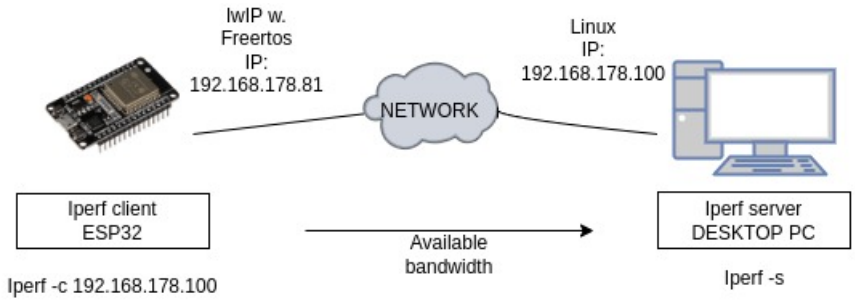


Figure 4.2: Iperfs werking met ESP32 als client

als in een afgeschermde (shielded) doos (samen met de access point) zijn uitgevoerd (Espressif, n.d.-b). Uit de resultaten zijn duidelijk af te leiden dat ruis en andere externe factoren een grote invloed hebben op de performance van de WiFi verbinding en daarmee ook de TCP/IP stack.

Type	Air in lab	Shielded box	Test tool
UDP RX	30 MBit/s	85 MBit/s	iperf example
UDP TX	30 MBit/s	75 MBit/s	iperf example
TCP RX	20 MBit/s	65 MBit/s	iperf example
TCP TX	20 MBit/s	75 MBit/s	iperf example

Figure 4.3: ESP32 Wi-Fi Throughput volgens Espressif

Dit onderzoek zal zich alleen richten op open lucht testen of zoals in het tabel hierboven benoemd Air in lab. Met de Wi-Fi access point naast de ESP32 en de PC bekabeld verbonden met de Wi-Fi accesspoint. Om de externe invloeden zoals ruis te beperken.

4.2.3 Testopstelling

Om het onderzoek repetitief onder dezelfde condities uit te kunnen voeren is er een testopstelling gemaakt. Deze testopstelling bestaat uit een ESP32, een Wi-Fi router en een Desktop PC met een variant van linux. De desktop pc is bedraad verbonden met de Wi-Fi router en de ESP32 is draadloos verbonden. De Wi-Fi router die gebruikt wordt is een Fritzbox 7390, dit modem heeft een theoretische maximale bandbreedte van 300Mbps (Wireless N), dit is voldoende voor de ESP32 die een theoretisch haalbare bandbreedte heeft van 138Mbps (Espressif, n.d.-a). Hieronder is de testopstelling geïllustreerd in een diagram:

Figure 4.4: Testopstelling schematisch uitgedrukt

4.2.4 Uitvoering

Het onderzoek wordt uitgevoerd door een script die automatisch de bandbreedte testen uitvoerd. Het script test ook de package sizes en window sizes. Dit doet het script door een nulmeting te doen met de standaard opties van Iperf. Vervolgens gaat het script de window size of package size aanpassen en vergelijken met de nulmeting. Het script verzamelt de gegevens en geeft een tabel wat meegenomen kan worden in dit rapport.

4.2.5 Testscript

Om de resultaten van het onderzoek te verzamelen is er een testscript geschreven in python. Dit script zal 3 testrondes uitvoeren en vervolgens een gemiddelde trekken uit de verzamelde gegevens. Deze gegevens zal het script in een text bestand zetten zodat het ingevoegd kan worden in een tabel. Het script wordt aan het einde van het verslag in het hoofdstuk bijlagen erbij gezet.

Chapter 5

Onderzoek

5.1 Uitgevoerd onderzoek

Er zijn drie soorten (deel)onderzoeken gedaan:
-Wat is de maximale bandbreedte van de ESP32 bij de standaard Iperf instellingen (Baseline).
-Welke window size geeft de hoogste bandbreedte
-Welke package size geeft de hoogste bandbreedte
De paragrafen van dit hoofdstuk geven de resultaten van elk onderzoek met een korte samenvatting van de uitvoering van het onderzoek.

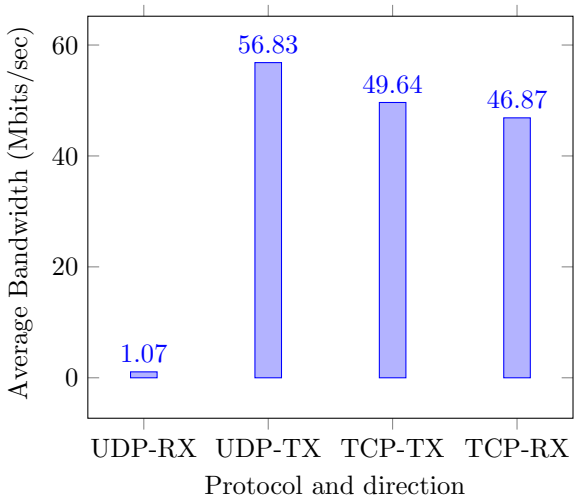
5.1.1 Baseline

In dit (deel)onderzoek is de bandbreedte onderzocht bij de standaard instellingen van Iperf.
De commando's gebruikt om deze resultaten te behalen zijn:

Testsoort	ESP32 Iperf commando	PC Iperf commando
UDP RX	iperf -u -s	iperf -u -c ESP32_IP
UDP TX	iperf -u -c PC_IP	iperf -u -s
TCP RX	iperf -s	iperf -c ESP32_IP
TCP TX	iperf -c PC_IP	iperf -s

De testopstelling is onveranderd en bestaat nog steeds uit een PC, ESP32 en een WiFi accespoint.
Elke test is 3 keer uitgevoerd en het gemiddelde resultaat is hieronder in een tabel en grafiek uitgezet.

Testsoort	Bandbreedte (Mbits/sec)
UDP RX	1.07
UDP TX	56.83
TCP TX	49.64
TCP RX	46.87



5.1.2 Window sizes

In dit (deel)onderzoek is de bandbreedte onderzocht bij de verschillende window sizes met behulp van Iperf.
De commando's gebruikt om deze resultaten te behalen zijn:

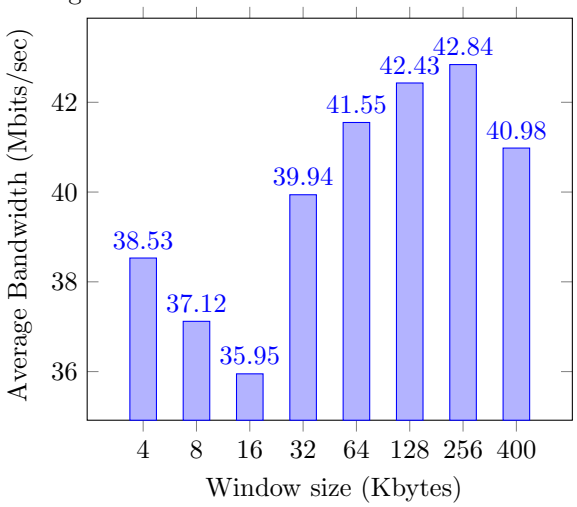
Testsoort	ESP32 Iperf commando	PC Iperf commando
Window size (ESP32 server)	iperf -s	iperf -c ESP32_IP -W Window_size
Window size (PC server)	iperf -c PC_IP	iperf -s -W Window_size

De testopstelling is onveranderd en bestaat nog steeds uit een PC, ESP32 en een WiFi accespoint.
Elke test is 3 keer uitgevoerd en de gemiddelde resultaten zijn hieronder in een tabel en grafiek uitgezet.

Window sizes test met ESP32 als server

Bij deze test wordt de bandbreedte gemeten met varierende window sizes waarbij de ESP32 als server en PC als client wordt gebruikt.

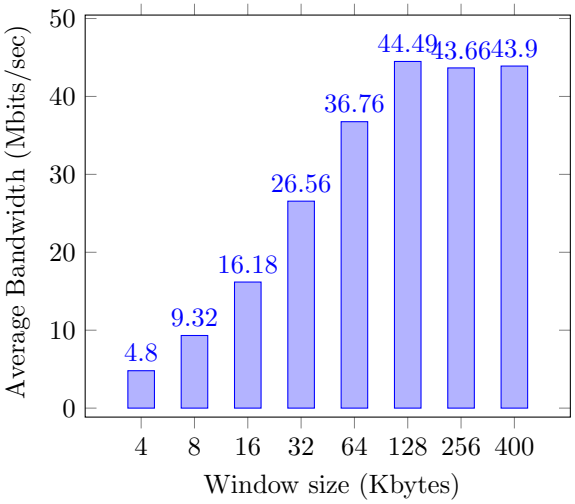
Window size(Kbytes)	Bandbreedte (Mbits/sec)
4 KB	38.53
8 KB	37.12
16 KB	35.95
32 KB	39.54
64 KB	41.55
128 KB	42.43
256 KB	42.84
400 KB	40.98



Window sizes test met PC als server

Bij deze test wordt de bandbreedte gemeten met varierende window sizes waarbij de ESP32 als client en PC als server wordt gebruikt.

Window size(Kbytes)	Bandbreedte (Mbits/sec)
4 KB	4.8
8 KB	9.32
16 KB	16.18
32 KB	26.56
64 KB	36.76
128 KB	44.49
256 KB	43.66
400 KB	43.9



5.1.3 Package sizes

In dit (deel)onderzoek is de bandbreedte onderzocht bij de verschillende package sizes met behulp van Iperf.
De commando's gebruikt om deze resultaten te behalen zijn:

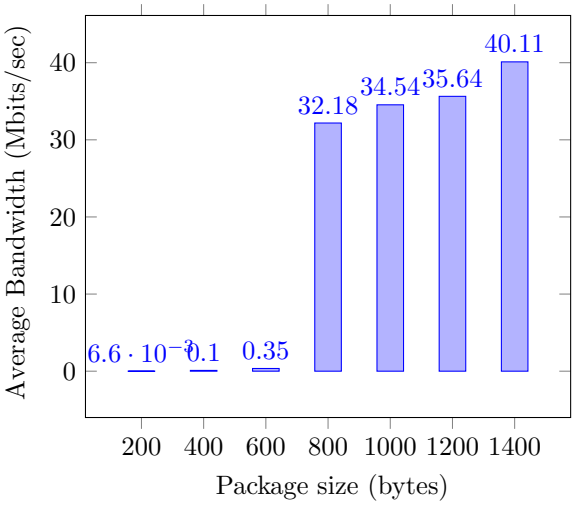
Testsoort	ESP32 Iperf commando	PC Iperf commando
Window size (ESP32 server)	iperf -s	iperf -c ESP32_IP -M MSS_size
Window size (PC server)	iperf -c PC_IP	iperf -s -M MSS_size

In de testen die uitgevoerd zijn, is de package size bepaald met behulp van de MSS grootte. MSS staat voor Max Segment Size en is het maximale formaat van het segment dat de NIC of PHY over het medium stuurt.
Of het ook daadwerkelijk het formaat is wat aangegeven wordt in de opties van het Iperf commando, is geen gegeven. De NIC of PHY kan afhankelijk van de fabrikant toch nog een andere MSS kiezen.
De testopstelling is onveranderd en bestaat nog steeds uit een PC, ESP32 en een WiFi accespoint.
Elke test is 3 keer uitgevoerd en de gemiddelde resultaten zijn hieronder in een tabel en grafiek uitgezet.

Package sizes test met ESP32 als server

Bij deze test wordt de bandbreedte gemeten met varierende package sizes waarbij de ESP32 als server en PC als client wordt gebruikt.

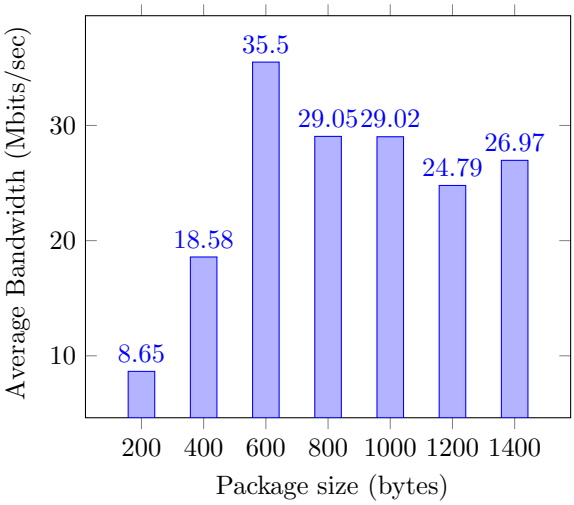
Package size(bytes)	Bandbreedte (Mbits/sec)
200 bytes (160 bytes MSS)	0.0066
400 bytes (360 bytes MSS)	0.1
600 bytes (560 bytes MSS)	0.35
800 bytes (760 bytes MSS)	32.18
1000 bytes (960 bytes MSS)	34.54
1200 bytes (1160 bytes MSS)	35.64
1400 bytes (1360 bytes MSS)	40.11



Package sizes test met ESP32 als client

Bij deze test wordt de bandbreedte gemeten met varierende package sizes waarbij de ESP32 als client en PC als server wordt gebruikt.

Package size(bytes)	Bandbreedte (Mbits/sec)
200 bytes (160 bytes MSS)	8.65
400 bytes (360 bytes MSS)	18.58
600 bytes (560 bytes MSS)	35.5
800 bytes (760 bytes MSS)	29.05
1000 bytes (960 bytes MSS)	29.02
1200 bytes (1160 bytes MSS)	24.79
1400 bytes (1360 bytes MSS)	26.97



Chapter 6

Resultaten en Conclusies

6.1 Samenvatting package size testen

De resultaten van de package size testen vertellen veel over de performance van de tcp/ip stack op de ESP32. Zo is er af te leiden dat bij het ontvangen een hoge package size wenselijk is. En bij lage package sizes de performance zwaar negatief wordt beinvloedt. De factoren die leiden tot deze negatieve beinvloeding zijn:

-Overhead door de grote hoeveelheid kleine paketten die elk een TCP en IP header meedraagt tot de eindbestemming.
-De tijd benodigt om de paketten te encapsuleren en decapsuleren.

6.1.1 Overhead

Om te berekenen hoeveel overhead er is bij een 200 bytes package size t.o.v. 1400 bytes met een data grootte van 5Kb wordt er gebruik gemaakt van dit diagram:

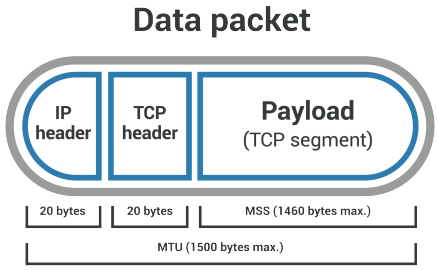


Figure 6.1: TCP/IP model

Uit dit diagram is duidelijk dat er per pakket al 40 bytes aan TCP en IP headers mee wordt gestuurd. Dan is hier al uit af te leiden dat voor het kleine pakket een overhead is van:

$$Overhead_{200bytes} = (\lceil \frac{N_{bytes}}{P_{size}} * 40) = (\frac{5000}{200} * 40) = 1000bytes$$

$$Overhead_{1400bytes} = (\lceil \frac{5000}{1400} * 40) = 160bytes$$

De minimale overhead bij het gebruik van een package size van 200 bytes bij het versturen van 5 Kbytes is 840 bytes t.o.v. de grootst geteste package size. Dit creert een significante hoeveelheid overhead op de verbinding. Deze headers en data moeten verzameld worden, ingepakt (geencapsuleerd), verzonden, uitgepakt(gedecapsuleerd) en (de data zelf) gereconstrueerd. Hoe meer van de pakketten hoe vaker deze taken opnieuw uitgevoerd moeten worden.

6.1.2 Encapsulatie en Decapsulatie

Encapsulatie is het process van het insluiten van een hogere laag protocol in een lagere laag protocol. Encapsulatie gebeurt in de lagen waarbij de informatie van de laag erboven ingepakt wordt in een nieuw pakket met de informatie van de huidige laag. Dit pakket wordt weer doorgegeven naar de laag eronder. Dit blijft zich herhalen tot het bij de fysieke laag komt. Nadat de fysieke laag gepasseerd is, wordt het pakket verstuurd over het medium.

Decapsulatie is het uitpakken van een hogere laags protocol uit een lagere laag protocol. Het kan ook wel omgekeerde encapsulatie genoemd worden.

Het telkens opnieuw inpakken van paketten neemt veel processor en geheugenkracht in beslag, dit effect speelt een rol in de testresultaten.

6.1.3 Optimale package size

Uit de testen is gebleken dat er een optimale package size is voor de ESP32. De meest optimale package size is 1400 bytes voor ontvangen en 600 bytes voor het verzenden van data.

6.2 Maximale bitrate

De maximale bitrate was ook een deelvraag. De hoogste bandbreedte van de resultaten waren bij de baseline testen met 56.83 Mbits/sec voor verzenden en 49.63 Mbits/sec voor ontvangen. Voor pakketten houd dit in dat het een bitrate van

$$56.83 \cdot 10^6$$

en

$$49.63 \cdot 10^6$$

bits per seconden. Voor de data die ontvangen en verzonden was:
MSS van 1400 bytes, vast header formaat van 40-bytes in 1 seconde:

$$Data_{send1s} = N_{bytes-(\lceil \frac{N_{bytes}}{P_{size}} * 40)} = 56.83{\cdot}10^6 - (\frac{56.83 \cdot 10^6}{1400} * 40) = 56789407 bytes$$

$$Data_{recv1s} = N_{bytes-(\lceil \frac{N_{bytes}}{P_{size}} * 40)} = 49.63{\cdot}10^6 - (\frac{49.63 \cdot 10^6}{1400} * 40) = 48212000 bytes$$

6.3 Voor welk doeleind is deze performance geschikt

De performance die behaald is in de testen is vrij hoog voor een embedded systeem. De verbinding is dusdanig hoog dat het videobellen, audiobellen, videoplayback en audioplayback ondersteund. Bestanden van een paar Mb kunnen ook zonder problemen in vrij korte tijd gedownload worden op deze verbinding.

Maar het is hoogstonwaarschijnlijk dat dit het doel was van het bedrijf Espressif toen de ESP32 op de markt werd gebracht. Het doeleind volgens de tech reference manual van de ESP32 (Espressif, n.d.-a) is de IOT markt. De IOT markt staat vooralsnog bekend als een markt waar het downloaden en uploaden van kleine hoeveelheden data de norm is (¡1 Kbyte normaal gesproken).

6.4 Algemene Conclusie

Het uitgevoerde onderzoek heeft een aantal dingen aan het licht gebracht. Zo is te concludereren dat de ESP32, ongeacht de package sizes, window sizes of protocol een hogere bandbreedte heeft met het verzenden (TX) dan het ontvangen (RX).
Bovendien is te concluderen dat de ESP32 optimale window en package sizes heeft waarbij de bandbreedte het hoogst is. De optimale window en package size is dan wel weer afhankelijk van verzenden of ontvangen. Bij ontvangen zijn grotere window en package sizes gewenst en bij verzenden kleinere.

Chapter 7

Bijlagen

7.1 Script

```
import serial
import subprocess
import time
import logging
import threading
#WiFi configuration
WiFi_SSID = 'TestAP'
WiFi_Password = "TestPassword"

# Needles for finding esp ip-addr in UART response
ip_search_str = "sta ip:"
ip_search_end_str = ", mask"

# Needles for finding bandwidth numbers in iperf output response
result_begin_str = "Bytes  "
result_end_str = "bits/sec"

# Setup command for initiating WiFi
Setup_CMD = "sta "+WiFi_SSID+" "+ WiFi_Password +"\r"

Initiate_server_CMD = "iperf -s -w "

Package_size_step = 10000

Read_buff_size = 200

def init_esp_iperf_server(s, window_size):
    s.write((Initiate_server_CMD+str(window_size)+"\r").encode())

def run_local_iperf_instance(server_ip_addr, options):
    proc = subprocess.Popen(["iperf -c "+server_ip_addr+" "+options
    ], stdout=subprocess.PIPE, shell=True)
    (output, err) = proc.communicate()
    outputstr = output.decode()
    return outputstr
```

```
35
36
37 def find_substring(src_str, begin_str, end_str):
38     index = src_str.index(begin_str) + len(begin_str)
39     index_stop = src_str.index(end_str)
40     return src_str[index:index_stop]
41
42 def init_esp_wifi(ser):
43     ser.write(Setup_CMD.encode())
44     print("Initializing ESP-Wifi..", end='')
45     read_buffer_str = ""
46     while(read_buffer_str.__contains__(", mask") == False):
47           read_buffer = ser.read(Read_buff_size)
48           read_buffer_str = read_buffer.decode('utf-8')
49           print('.', end='')
50     time.sleep(2) #Wait for the ESP32 to initialize wifi connection
       , takes approximately 2 secs
51     return find_substring(read_buffer_str, ip_search_str,
       ip_search_end_str)
52
53 def reset_esp32(ser):
54     ser.write("restart\r".encode())
55     ser.flushOutput()
56     time.sleep(5) #Wait for the ESP32 to restart, takes
       approximately 5 secs
57
58 def main():
59     ser = serial.Serial(
60             # Serial Port to read the data from
61             port='/dev/ttyUSB0',
62
63             #Rate at which the information is shared to the
       communication channel
64             baudrate = 115200,
65
66             #Applying Parity Checking (none in this case)
67             parity=serial.PARITY_NONE,
68
69            # Pattern of Bits to be read
70             stopbits=serial.STOPBITS_ONE,
71
72             # Total number of bits to be read
73             bytesize=serial.EIGHTBITS,
74             # Number of serial commands to accept before timing out
75             timeout=0
76     )
77     file1 = open("log.txt", "w")
78     for step in range(1,11):
79         time.sleep(.1)
80         reset_esp32(ser)
81         server_ip_addr = init_esp_wifi(ser)
82         print(server_ip_addr)
83         package_size = Package_size_step*step
84         init_esp_iperf_server(ser, package_size)
85         outputstr = run_local_iperf_instance(server_ip_addr, " -w "
       +str(package_size)+" -t 5")
86         print(outputstr)
```

```
87          result = find_substring(outputstr, result_begin_str,
    result_end_str)
88          file1.write(result+"\n")
89          file1.write(str(package_size)+"\n")
90      file1.close()
91
92
93  if __name__=="__main__":
94      main()
```

Bronnen

David Stepner, N. R., & Hui, D. (1999). Embedded application design using a real-time os. http://www.cecs.uci.edu/~papers/compendium94-03/papers/1999/dac99/pdffiles/09_2.pdf

Dunkels, A. (2001). *Design and implementation of the lwip tcp/ip stack*. https://www.artila.com/download/RIO/RIO-2010PG/lwip.pdf

Dunkels, A. (2003). *Full tcp/ip for 8-bit architectures*. https://www.usenix.org/legacy/publications/library/proceedings/mobisys03/tech/full_papers/dunkels/dunkels.pdf

Espressif. (n.d.-a). *Esp32 technical reference manual*. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

Espressif. (n.d.-b). *Esp32 wi-fi throughput*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html#esp32-wi-fi-throughput

Espressif. (n.d.-c). *Lwip*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/lwip.html

Jacob, B. L. (2000). *An evaluation of embedded system behavior using full-system software emulation*.

M. K. McKusick, M. J. K., K. Bostic, & Quarterman, J. S. (1996). *The design and implementation of the 4.4 bsd operating system*. University of Michigan: Addison-Wesley.

Majkovski, M. (n.d.). Why we use the linux kernel's tcp stack. https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/

Wang, W. (2021). Finding and exploiting vulnerabilities in embedded tcp/ip stacks. *Master of Science in Technology Thesis*, 17–20. https://www.utupub.fi/bitstream/handle/10024/152445/Thesis-Wenhui-Wang.pdf