**NU Module II Final Assignment Report**

Renske Schram

PROTOCOL

De Client en Server maken gebruik van het UDP Stop-and-Wait ARQ. Het plan tijdens deze twee weken was om eerst een betrouwbaar systeem te maken en daarna efficiënter te maken met o.a. een Go Back N ARQ protocol. Dat vond ik heel lastig omdat ik merkte dat ik aan het begin meteen goed en met veel extraatjes wilde opbouwen, maar daardoor ook al problemen creëerde in de opzet die nog niet aan de orde waren. Uiteindelijk heb ik er in de laatste anderhalve dag ervoor gekozen om verder te testen met het instabiele netwerk, tests uit te breiden en de code beter te structureren, omdat het stiekem toch meer tijd en structurele aanpassingen kostte dan ik had ingepland om mijn ene systeem om te zetten naar het Go Back N AQR protocol en goed te kunnen testen. Een deel van de code die ik hiervoor heb geschreven en niet gebruikt heb ik weer weggehaald en een deel staat er nog (waardoor sommige functies/variabelen misschien iets omslachtig lijken, maar dat komt dus voort uit wel nog de stap hebben willen maken naar Go Back N ARQ). Het protocol is verder afhankelijk van de Flags in de Header. Elk valide ontvangen berichtje wordt door de Host beantwoord met een ACK-flagged Packet, waarna de zender het volgende pakketje kan sturen. Verder heb ik de flag HELLO toegepast om de ontvanger te laten weten dat dit een initiatiepakketje is voor een bepaalde actie en FIN om te laten weten dat dit het laatste pakketje is binnen de actie. Verder gevend de LIST, GET en DATA flags aan dat het een aanvraag voor respectievelijk een directory list, download of upload vanuit de zender is. Ik heb aan het eind ook nog een ERROR flag toegepast zodat eventuele ERRORS kunnen worden doorgegeven.

NETWORKING

Ik heb het zo opgezet dat de Client en Server zoveel mogelijk met elkaar overeenkomen, in principe moeten ze natuurlijk ook beiden hetzelfde netwerkprotocol volgen om correct met elkaar te communiceren, waarbij Packets hetzelfde moeten worden afgehandeld. Een verschil wat ik heb toegepast is dat de Server continu draait en altijd reageert op een berichtje van de Client, waarbij de Client juist (via de TUI) de mogelijkheid heeft een bepaalde actie (of service zoals ik qua termen heb aangehouden) kan initiëren). Om dit verschil tot stand te brengen heeft de Client t.o.v. de Server de extra functies om een upload, download en list aanvraag te starten. Hiermee maakt de Client kant Packets aan waarmee het proces wordt afgetrapt en de Server kan reageren (want in principe is een download voor de Client kant natuurlijk een upload voor de Server kant en vice versa). Om het in eerste instantie simpel te houden, heb ik ervoor gekozen dat de Client bijhoudt wanneer deze de laatste Ack heeft ontvangen/verzonden om zo de service af te ronden en een nieuwe actie/input kan ontvangen (in plaats van bijvoorbeeld met een Thread te werken, waardoor de Client helemaal gelijk aan de Server zou kunnen blijven en bijv. geen lastAck hoeft te bepalen). Verder volgen beiden Hosts hetzelfde principe van drie “lagen” van Packet verwerking:

1. *(Abstract)Host (extended door Client/Server)*

Ontvangen: DatagramPackets komen binnen bij de AbstractHost (extended door een Client/Server). Deze controleert of de Packet valide is (Checksum verificatie), zo ja dan wordt deze direct acknowledged. DatagramPacket wordt omgezet in een normale Packet (dus de InetAdress, port etc. zit alleen in deze laag). Timer van ontvangen Packet wordt zo nodig gecanceld en als het de juiste Packet is in volgende van ontvangen, wordt deze doorgestuurd naar de volgende laag. Is het niet de juiste in volgorde (zoals kan voorkomen met een sliding window), kan deze tijdelijk worden opgeslagen tot deze aan de beurt is.

Zenden: Packets die verzonden moeten, komen weer bij de AbstractHost laag. Deze maakt er weer een DatagramPacket voor de juiste bestemming, zet de Checksum over het uiteindelijke pakketje en zet zo nodig een Timer.

Op deze manier deze laag verantwoordelijk voor zenden/ontvangen en correct houden bij instabiel netwerk middels gebruik van timer, checksum en evt. volgorde van afleveren van Packets aan de volgende laag middels de receivingWindow.

1. *ServiceHandler*

Correcte Packet komt binnen en afhankelijk van de Flags in de Header wordt deze afgehandeld. Moet er een actie in gang gezet moeten worden, dan wordt de Packet en inhoud doorgegeven aan een van de volgende lagen. Anders kan deze ServiceHandler-laag, met behulp van de PacketBuilder en diens static methodes om standaard Packets aan te maken, direct de juiste response Packet teruggeven aan de AbstractHost. Met het teruggeven van de Packets houdt deze laag ook de sendingWindow bij om te bepalen hoeveel Packets er kunnen worden opgehaald uit de diepere lagen, mocht dat relevant zijn (voornamelijk bij uploaden). Voor elke Packet die wordt teruggegeven naar de hogere laag, wordt de juiste Ack nummer in de Header verwerkt en sendingWindow geüpdatete.

Op deze manier is deze laag verantwoordelijk voor het handelen met de Packets op basis van de Flags: met de Payload van de Packets diepere lagen aansturen, juiste Packets ophalen en met de juiste Ack nummer teruggeven aan de Host om verzonden te worden.

*3a. FileLoader*

Deze laag is verantwoordelijk voor het opknippen van een bestaand bestand in kleinere stukken byte array die als Payload kan worden meegegeven aan een Packet. Als de FileLoader geïnitieerd is, dan heeft deze een source path van de File die deze aan het verwerken is, houdt bij waar in de file deze bezig is middels de offsetPointer en wanneer de FileLoader klaar is. Op aanvraag kan het volgende stukje File worden meegegeven als Payload in een Packet. Wanneer het hele bestand is verwerkt en meegegeven, reset de FileLoader zichzelf, zodat deze weer opnieuw geïnitieerd kan worden bij een nieuwe upload actie. De FileLoader kan alleen worden geïnitieerd bij als het te uploaden bestand ook bestaat, anders wordt een ERROR Packet teruggegeven en verstuurd om de upload af te breken.

*3b. FileBuffer*

Deze laag is verantwoordelijk voor het opbouwen van de losse Payload tot een File. Met het ontvangen initiatie pakketje wordt een byte buffer aangemaakt, waar met elk ontvangen pakketje de Payload op de juiste plaats in deze buffer wordt gekopieerd. In dit initiatie pakketje zit ook direct de directory met bestandsnaam/type om correct te kunnen bouwen. Wanneer de buffer vol is en het laatste pakketje in de buffer zit, wordt deze omgeschreven naar de correcte File format en naam, waarna de Buffer zichzelf weer reset, klaar om een volgende File te bouwen. Bestanden overschrijven andere bestanden om deze zodoende te kunnen vervangen. Bestaat de meegegeven directory nog niet, dan wordt deze aangemaakt.

Voordeel van deze opzet is dat alle lagen echt los van elkaar functioneren en functionaliteiten of juist fouten echt in een eigen laag al worden afgevangen. Met de begin opzet van deze indeling kon ik de classes hierdoor ook relatief klein, compact en duidelijk houden, dat is tegen het einde wel iets meer geworden omdat er nog wat losse functies nodig waren. Zo had het verwerken van de meer “bericht achtige” pakketjes als het printen van de List of error misschien ook los gekund, omdat dat net als het loaden/bufferen van een file een derde manier is om pakketjes af te handelen.

PACKET

*(Datagram)Packet* Voor het netwerken tussen de Client en de Server heb ik de DatagramPacket toegepast en gebruik gemaakt van de variabelen length, address en port. *Length* om een binnengekomen DatagramPacket in de buffer te kunnen omzetten naar een Packet met de juiste lengte, en *address* en *port* om de juiste andere Host te bereiken. De Host laag stript deze DatagramPacket laag en creëert hiervan een Packet object die verder de basis vormt vanuit waar de vervolgacties in gang worden gezet. De Packet bestaat uit een Header en Payload.

*Payload* Het idee is dat de Payload puur bytearray data bevat met of een deel van een File of een bytearray verkregen vanuit een String met informatie. Er zijn dan twee Payload constructors, één per soort Payload. Payload heeft lengte als eigenschap en ik heb ook de offsetPointer als variabele meegegeven. Deze zit niet in de bytearray van de Payload zelf, maar vind ik wel een eigenschap van de data van de Payload. De offsetPointer zit wel in de bytearray van de Header. Ik heb ervoor gekozen om de String data in de Payload te splitsen middels een “~”, nog voortkomend uit eerdere gebruiken. Achteraf gezien met het werken met Files en directories is dat misschien niet de meest handige keuze, aangezien de tilde wel eens gebruikt wordt met navigeren naar het juiste bestand/folder, maar het geeft binnen de huidige functionaliteiten geen probleem.

*Header* De Header bestaat alleen uit een bytearray. In die bytearray zitten de twee variabelen payloadlength en offsetPointer opgeslagen. Verder bevat deze ook een Ack. nummer en Seq. nummer (alvast voor de uitwerking van de sliding window, uiteindelijk dus niet in de laatste versie toegepast), checksum voor data integriteit van de hele Packet en een byte met Flags. Middels deze Flags weet de ontvangende host wat het doel is van dit pakketje, zo wordt altijd begonnen met een HELLO-pakketje (en GET/DATA/LIST voor de functie) en geëindigd met een FIN om aan te geven dat dit inderdaad het laatste pakketje is van deze actie.

UTILS

Om de andere classes zo schoon mogelijk te houden heb ik een aantal util Classes gemaakt met voornamelijk gestandaardiseerde static functies en variabelen, zoals: *Checksum* met de een manier om over een Packet de Checksum te berekenen en te verifiëren of de Packet de juiste checksum heeft (en dus integer is). *LoggingHandler* om system.err.prints om te leiden naar een host.log file, zodat de Client console schoon blijft en hierin de status notificaties te zien zijn en andere interresante informatie voor debuggen (zoals timed out Packets en weergeven van de sliding window) in de host.log terug te vinden is. *PacketBuilder is* een class met static functies die een ingevulde Packet teruggeven op basis van enkele inputs.

*Parameters*

Ik heb een los parameter bestandje aangemaakt om makkelijk bij veelgebruikte parameters te komen, zoals de parameters voor de timer en standaard directories. Ik ben ook initieel begonnen met de max packet size als vaste variabele voor de client en Server kant om mee te experimenteren. Voor de eerste opzet werkt dat prima, maar nu ik heb getest met het instabiele netwerk zou ik in het vervolg meteen implementeren dat functies meer gebruik maken van de size die wordt meegegeven in de Packet zelf zodat flexibeler gereageerd kan worden op de grootte van pakketjes die het netwerk op dat moment aankan. Met een betrouwbaar netwerk en grote bandbreedte helpt een grote packetsize om met weinig stappen snel de data over te brengen, maar als deze omstandigheden minder zijn, dan kan dit snel belemmeren. Na verschillende keren testen met een instabiel netwerk geeft 8192 bytes als Packet grootte voorlopig het snelste resultaat. Een andere variabele die daarin sterk meespeelt is de timout duration, dus hoelang het duurt voor een packet opnieuw wordt verzonden. Omdat het uiteindelijk toegepaste protocol een stop and wait protocol is, kan de time out het beste worden gezet op de round trip time (RTT) plus een beetje speling om zo zo min mogelijk onnodige herzendingen te sturen. RTT is in dit geval vooral afhankelijk van de delay die het instabiele netwerk heeft en is de delay ook voornamelijk bepalend voor de snelheid waarmee de uploads en downloads plaatsvinden. Nu liggen deze parameters vooraf vast, in het vervolg zou het beter zijn om deze bij de host te bepalen en aan te passen op basis van de snelheid waarmee pakketten binnen komen en het aantal dat verloren gaat met verschillende resource allocation/congestion control technieken, zoals Additive Increase, Multiplicative Decrease (AIMD).

Met ideale omstandigheden en zonder het packet loss en network delays toe te voegen aan het netwerk lukt het met de huidige opzet om een tiny.pdf te sturen in 0.067 sec en ophalen in 0.021 sec, medium.pdf te sturen in 0.366 sec en op te halen in 0.277 sec en large.pdf te sturen in 25.943 seconden en op te halen in 14.79 sec, alles zonder verlies/delay van pakketjes. Zodra ik er een delay van 1s en packetloss van 15% aan toevoeg, valt te zien dat (1) het systeem wel blijft werken zonder fouten, maar (2) het flink verslechterd in prestatie. Uploaden/downloaden van tiny.pdf kost 4.19 sec (2 Packets lost/delayed), medium.pdf 62.294 sec (16 Packets lost/delayed) upload en 82.234 sec download (40 Packets lost/delayed) en large.pdf… dat is niet heel fraai met veel packet loss/delay (dus herzenden) per 3 a 4 pakketjes en bijna een uur aan laadtijd. Packet size groter maakt help voor het zenden wel om de tijd te verminderen (dan hoeft de Pi alleen te reageren met kleine Ack Packets), alleen bij het downloaden van een File naar de Client wordt het systeem minder betrouwbaar omdat soms wel teveel achter elkaar een Packet niet wordt verstuurd vanuit de Server door de overbelasting. Hiervoor zou dan echt de oplossing zijn om AIMD toe te passen en aan het netwerk te anticiperen met de Packet size en natuurlijk wel sliding window om minder last te hebben van de delays.