Karelia-ammattikorkeakoulu
Tradenomi (AMK), Tietojenkäsittely

PCAP-rajapinnan hyödyntäminen verkkoliikenteen hallinnassa

Niko Huuskonen

Opinnäytetyö, lokakuu 2024

www.karelia.fi

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Lokakuu 2024
Tietojenkäsittelytieteiden koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä(t)
Niko Huuskonen

Nimeke
PCAP-rajapinnan hyödyntäminen verkkoliikenteen hallinnassa

Toimeksiantaja
Nolwenture Ltd.

Tiivistelmä

Tässä opinnäytetyössä toteutetaan kehitysprosessi ohjelmistolle, joka oli suunniteltu keräämään ja purkamaan verkkodataa ja toimimaan pohjana jatkokehittämiselle. Verkkodata saadaan ohjelmistolle määritellystä verkkoportista käyttäen apuna PCAP-rajapintaa ja tarjoamaan käsitellyn datan ohjelmistorajapintana jatkokehittämistä ja datan jatkokäsittelyä varten.

Ohjelmiston toteuttamisessa käytettiin Typescript-ohjelmointikieltä, PCAP-rajapintaa ja Docker-kontitusohjelmistoa. Ohjelmistokokonaisuudessa oli painotettu modulaarisuutta ja täten muokattavuutta, jotta se olisi mahdollisimman helppo laajentaa erilaisiin käyttötarkoituksiin. Ohjelmiston kehitysprosessi tapahtui ydintoiminnallisuudesta ylöspäin, verkkokerros kerrallaan ylöspäin OSI-mallin mukaisesti. Itse verkkodata tulee suoraan PCAP-rajapinnasta, joka puolestaan kaappaa datan verkkoportista, jota se kuuntelee. Jokaisen verkkokerroksen datankäsittelijä on kykenevä toimimaan itsenäisesti, eli datankäsittelijä on ikään kuin itsenäinen moduuli. Datankäsittelijöille suunniteltiin ja toteutettiin verkkokerroksittain tarvittavat yksikkötestit. Eri kerroksien datanpurkajat puolestaan ketjutettiin yhteen isomman kokonaisuuden sisälle ja tämä kokonaisuus puolestaan tarjoaa yhden ison tapahtumapohjaisen kehitysrajapinnan, jotta ohjelmistopohjaa käyttävä kehittäjä pystyisi hyödyntämään helppoa jatkokehittämistä varten.

Ohjelmistokokonaisuus onnistui hyvin, verkkodata luetaan ja prosessoidaan nopeasti ja data on helposti tulkittavassa ja tyyppivarmassa muodossa. Lisäksi ohjelmiston laajentaminen on helppoa. Ohjelmisto vastaa sille asetettuja tavoitteita, joskin projektikokonaisuuden ajankäyttö venyi suuremmaksi, kuin mitä sille oli alun perin määritelty.

Kieli
suomi

Sivuja 56

Asiasanat
Ohjelmistokehitys, tietoverkot, modulaarisuus

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
October 2024
Degree Programme in Computer Science

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600 (switchboard)

Author (s)
Niko Huuskonen

Title
Utilization of PCAP-API in Network Flow Management

Commissioned by
Nolwenture Ltd.

Abstract

In this thesis, a software development process was carried out for a software that was designed to extract and decrypt data from a specified network port with the help of PCAP and then provide the processed data as an application programming interface for further development and usage of the data.

The software was implemented using the Typescript programming language, PCAP-API and the Docker framework. The software package was designed to be very modular and thus customizable, to be as easy as possible to extend and adapt for different use cases. The software development process was carried out from the core functionality upwards, one network layer at a time, following the OSI model. The network traffic data itself comes directly from the PCAP interface, which in turn gets it by capturing the traffic from the network port it is assigned to listen to. Each data handler was designed to take care of a single network layer and can operate independently, i.e., the data handler is a kind of stand-alone module. The necessary unit tests for each data handler were designed and implemented. The data handlers of the different network layers were in turn chained together within a larger entity, which in turn provides a single large event-based development interface for easy further development by the programmer using the software platform.

In the end the software project turned out well: The traffic data is captured and processed quickly, and the data is both easy to read and in a type-safe format and expandability is easy. Overall, the software meets the requirements of the commission, although the project schedule did stretch beyond what was originally specified.

Language
Finnish

Pages 56

Keywords
Software development, information networks, modularity

Sisältö

Sanasto

API

API, eli ”application programming interface” tarkoittaa ohjelmointirajapintaa. Ohjelmointirajapinnan tehtävänä on mahdollistaa kahden tai useamman ohjelmiston tai ohjelmistokomponentin välinen kommunikaatio. Kommunikaatiokanavat ovat yleensä tyyliltään avoimia ja dokumentoituja, jotta niiden käyttäminen olisi mahdollisimman helppoa.

Asynkronisuus

Kun yleisesti puhutaan asynkronisuudesta, tarkoitetaan, että asiat tapahtuvat itsenäisesti, eivätkä vaikuta toisiinsa. Ohjelmoinnin puolella asynkronisuudella tarkoitetaan operaatioita, jotka eivät estä toisten operaatioiden suorittamista, eli tämä mahdollistaa monen operaation samanaikaisen suorituksen. Tämä on äärimmäisen tärkeää ympäristöissä, jossa funktiot vastaavat asioista ja operaatioista, joiden suoritustarvetta ei voida ennakoida. Tällaisia ympäristöjä ovat esimerkiksi tietoverkot, HTTP-palvelimet ja tietokoneen IO:sta vastaavat ajurit.

Datalinkkikerros

Datalinkkikerros, eli data link layer on verkkokerrosta tason alempana, eli OSI-mallin toinen kerros. Tässä kerroksessa viestintä tapahtuu yhdestä laitteesta toiseen, eli kerroksen viestintä on pisteestä pisteeseen. Varsinaista verkkorakennetta ei ole, tästä vastaa verkkokerros. Datalinkkikerroksessa laitteet erotellaan toinen toisistaan MAC-osoitteiden avulla.

Dekoodaus

Tietotekniikan ja ohjelmoinnin kontekstissa dekoodaamisella (eng. decoding) tarkoitetaan datan purkamista formaatista toiseen. Yleensä datanpurulla muutetaan kompaktia, pakattua dataa luettavampaan ja käytettävämpään muotoon. Dekoodattu data on usein enkoodattua dataa suurempaa, jonka takia datan dekoodaus tehdään vasta, kun dataa tarvitaan. Dekoodaaminen, kuin myös enkoodaaminen on olennainen osa etenkin tiedonsiirrossa.

Enkoodaus

Enkoodaus (eng. encoding) on vastakohta dekoodaamiselle, eli data pakataan suuremmasta, useimmiten luettavammasta muodosta tiiviimmäksi, jotta sen välittäminen ja/tai varastointi olisi tehokkaampaa. Jotta enkoodattua dataa voidaan käyttää, täytyy se dekoodata takaisin käytettävämpään ja luettavampaan muotoon.

Ethernet-kehys

Ethernet-kehys, eli Ethernet Frame on yksittäinen datapaketti datalinkkikerroksessa. Ethernet-kehyksen otsakkeet pitävät sisällään datalinkkikerroksen kommunikaatiolle vaadittavat tiedot, kuten MAC-osoitteet, kehyksen datan tyypin ja mahdollisen VLAN-datan. Ethernet-kehykset ovat OSI-mallin alin varsinaisen digitaalisen datarakenteen omaavia tietopaketteja, jossa data on paloiteltu erillisiksi osiksi, tässä tapauksissa kehyksiksi. Alemman kerroksen, fyysisen kerroksen, data on puhdasta raakaa bittivirtaa.

EtherType

EtherType on pakollinen kahden oktetin kokoinen tietokenttä Ethernet-kehyksen otsakkeessa. Sen tehtävänä on indikoida, mitä protokollaa kapseloitu data on ja vastapuolen datalinkkikerros käyttää tätä tietoa, jotta se osaa purkaa datan oikein.

Git-versionhallinta

Git-versionhallinta on pääasiassa hajautettuun ohjelmistonkehitykseen suunnattu versionhallintajärjestelmä, joka pitää kirjaa lähdekoodiin kehityksen aikana tapahtuvista muutoksista. Tämä mahdollistaa helpon tavan usealle kehittäjälle työskennellä saman lähdekoodin parissa, hallita ohjelmiston versiointia, sekä korjata mahdollisia koodikonflikteja. Gitin pääominaisuudet ovat tietovarastot (repository), versiohaarat (branch) sekä solmut (commit).

IDE

IDE, lyhenne sanoille ”integrated development environment”, tarkoittaa ohjelmaa, joka tarjoaa kattavat työkalut ohjelmistojen kehittämiseen, kehitysympäristöä. Yleensä kehitysympäristö pitää sisällään itse varsinaisen lähdekoodieditorin, rakennustyökalun eli compilerin sekä virheenkorjaustyökaluja kehitettävän ohjelman tai ohjelmiston vianetsintään sekä testaukseen. Lisäksi kehitysympäristöt tyypillisesti sisältävät ohjelmointityötä tehostavia sekä helpottavia ominaisuuksia, kuten automaattisen täytön, lähdekoodin osien väritys, sekä koodianalysoijan, joka ilmoittaa mahdollisista syntaksi- sekä toiminnallisuusvirheistä jo koodia kirjoittaessa. Yleisesti käytössä olevia kehitysympäristöjä ovat esimerkiksi Visual Studio Code, Visual Studio, IntelliJ sekä Eclipse.

IP-osoite

IP-osoitteet ovat numerosarjoja, joita käytetään tietoverkkoon kytkettyjen laitteiden yksilöimiseen. Lyhenne IP tulee englannin kielen sanoista ”internet protocol”. Toisin kuin laiteverkkokerroksen MAC-osoitteet, jotka mahdollistavat vain suoran tiedonsiirron kahden laitteen välillä, IP-osoitteet mahdollistavat epäsuoran tiedonkulun laitteiden välillä käyttäen verkon ”välissä” olevia laitteita välittämään dataa eteenpäin. IP-osoitteita on käytössä pääasiassa kahdessa standardissa: IPv4 ja IPv6.

IPv4

IPv4-osoitteet ovat 32 bitin kokoisia osoitteita, jotka ovat jaettu neljään 8 bitin suuruiseen osaan. IPv4 on IPv6 nähden vanhempi, mutta vielä tänäkin päivänä laajemmin käytössä oleva standardi. Teoreettinen maksimimäärä yksilöllisille IPv4-osoitteille on 2^32, eli 4 294 967 296. Tämä on osoittautunut merkittäväksi rajoittavaksi tekijäksi IPv4-standardissa, jonka vuoksi tietoverkot ovat siirtymässä IPv6-standardiin. IPv4-verkkopaketin otsikoiden koko vaihtelee 20 ja 60 tavun välillä, riippuen otsikoiden sisällöstä. Esimerkki IPv4-osoitteesta on 192.168.1.1.

IPv6

IPv6-osoitteet ovat IPv4-osoitteista poiketen 128-bittisiä ja ne ovat jaoteltavissa 16 bitin kokoisiin osioihin. Suuren bittimäärän vuoksi IPv6 ei kärsi IPv4:lle tyypillisestä vähäisten osoitteiden määrästä, koska teoreettinen osoitteiden määrä on 2^128. Myös IPv4-osoitteista poiketen IPv6-osoitteissa osiot erotellaan toisistaan kaksoispisteellä (:) pisteen (.) sijaan ja IPv6-verkkopaketin otsikot ovat aina yhtä pitkät (40 tavua), sisällöstä riippumatta. Esimerkki IPv6 osoitteesta on 2001:db8::ff00:42:8329.

Lähdekoodi

Lähdekoodi, englanniksi ”source code” on ihmisen luettavissa tekstimuodossa oleva koodi, josta ohjelma rakentuu, eli kääntyy tietokoneen tulkittavaan ja suoritettavaan muotoon. Ohjelmistot rakennetaan lähdekoodilla ennalta valitulla ohjelmointikielellä. Versionhallintaa käytetään lähdekoodin hallintaan.

MAC-osoite

MAC-osoitteet ovat verkkokerrosta (vrt. IP-verkko) alemman ethernet-kerroksen yksilöiviä osoitteita. Samoin kuin MAC on lyhenne sanoille ”media access control”. Toisin kuin IP-osoitteet, jotka ovat täysin virtuaalisia ja täten dynaamisia, MAC-osoitteet ovat yleensä fyysisesti kirjoitettu verkkokorttiin, mutta sen voi silti muuttaa ohjelmistossa jälkikäteenkin.

Ohjelmointikieli

Ohjelmointikieli, englanniksi ”programming language” on kieli, jolla lähdekoodi kirjoitetaan. Eri ohjelmointikielet kääntyvät kukin niille tarkoitetulla koodikääntäjällä joko binäärimuotoon tai koodi kääntyy reaaliaikaisesti sitä suoritettaessa virtuaalikoneessa JIT-periaatetta (just in time) noudattaen, eli lähdekoodi kääntyy tietokoneelle ymmärrettävään muotoon suoritusvaiheessa. Tämä takaa omat etuutensa, kuten dynaamisen modulaarisuuden, mutta vastaavasti tämän tapaisien ohjelmointikielien suorituskyky ei ole yleensä yhtä nopea. Esimerkkejä etukäteen kääntyvistä ohjelmointikielistä ovat C, C++, Rust ja Go sekä JIT-kielistä Javascript, Python ja Typescript.

OSI-malli

OSI-malli on yksinkertaistettu pino tiedonsiirron eri tasoista. Tässä mallissa verkkoliikenne jaotellaan 7 kerrokseen, joista kukin kerros tekee tietyn asian ja kapseloi varsinaisen datan. Alempi kerros kapseloi aina ylemmän kerroksen datan ja vastaavasti purkaa oman kapseloinnin antaessa dataa takaisin ylemmällä kerrokselle.

PCAP

PCAP, lyhenne sanoista ”packet capture”, on ohjelmointirajapinta (API) tietoverkkoliikenteen kaappaamista varten ilman, että se vaikuttaa alkuperäiseen verkkoliikenteeseen. Itse API:n nimi ei kuitenkaan ole PCAP, vaan Unix-järjestelmissä libpcap ja Windows-järjestelmissä Npcap. Molemmat rajapinnat ovat toteutettu C-ohjelmointikielellä parhaan suorituskyvyn takaamiseksi ja tarjoavat saman toiminnallisuuden.

Solmu (Git)

Solmut ovat Git-versionhallinan pienimpiä elementtejä. Kukin solmu on eräänlainen aikakapseli siitä, miltä ohjelmiston lähdekoodi näytti kyseisellä hetkellä. Jokaisella solmulla on oma uniikki tunnuksensa, jolla solmut ovat eroteltavissa toisistaan. Lisäksi solmua luodessa voidaan kirjoittaa solmun otsikko, sekä kommentti siitä, mitä solmu pitää sisällään.

Suorituskontti

Suorituskontti, englanniksi ”container runner”, on työkalu konttien hallintaan ja suorittamiseen. Kontit ovat kevyitä, modulaarisia ja itsenäisesti toimivia ohjelmistopaketteja, jotka kapseloivat niissä suoritettavan sovelluksen ja sen riippuvuudet, mikä takaa luotettavan ja samantasoisen toiminnallisuuden ajoympäristöstä riippumatta, muutamia poikkeuksia lukuunottamatta. Esimerkkejä tällaisista kontitusohjelmistoista ovat Docker, OpenVZ, LXC ja Podman.

Synkronisuus

Synkronisuus on eräänlainen vastakohta asynkronisuudelle, eli koodia ajetaan järjestelmällisesti operaatio kerrallaan. Muut operaatiot odottavat edeltävän

operaation valmistumista, eli ajettava operaatio ”estää” muita operaatioita suorittumasta. Suorituksen estämisellä on haittansa, kuten ohjelman hidas toiminta ja joutoaika, jolloin ohjelman muiden osien toiminta pysähtyy. Kuitenkin joskus tämä on välttämätöntä kriittisissä, mutta ei aikaan sidonnaisissa operaatioissa, koska synkronisen koodin toiminta on ennakoitavissa ja sen vianetsintä on huomattavasti asynkronista koodia helpompaa.

Tietovarasto (Git)

Tietovarastot ovat kokoelmia, jotka sisältävät ohjelmakomponentin tai ohjelmiston lähdekoodin ja versionhallinnan solmut sekä haarat. Tietovarasto on ikään kuin säiliö, joka varastoi kaiken kyseiseen koodiin liittyvän versionhallinnan sekä itse lähdekoodin osalta.

Verkkokerros

Verkkokerros, englanniksi internet layer, on OSI-mallin kolmas kerros. Esimerkiksi IP-verkot toimivat tässä kerroksessa. Verkkokerros vastaa laitteiden välisestä kommunikaatiosta riippumatta siitä, kuinka monen ”hypyn” päässä kohdelaite on. Verkkokerros siis muodostaa eräänlaisen laitteiden välisen viestintäverkon, mistä tuleekin itse kerroksen nimi.

Verkkopaketti

Verkkopaketti (network packet) on tietoverkossa, pääasiassa IP-verkossa, kulkeva datapaketti. Paketin sisältö alkaa aina standardin määrittelemillä otsikoilla, jotka kertovat paketin metadatan, kuten paketin koon, lähtöosoitteen ja kohdeosoitteen. Paketin sisältö on kapseloitu ja sisällön käsittelystä vastaa paketin lähettävä ja vastaanottava taho. Pakettia välittävät verkkolaitteet eivät lähtökohtaisesti muokkaa sisältöä.

Verkkoportti

Verkkoportti, englanniksi ”network interface”, on joko fyysinen tai virtuaalinen kanava, jonka avulla joko tietokone, virtuaalitietokone tai esimerkiksi Docker-kontti yhdistetään tietoverkkoon. Virtuaaliset verkkoportit ovat ohjelmistollisia,

mikä mahdollistaa niiden helpon reitittämisen ja kytkemisen toisiinsa, sekä virtuaalisten lähiverkkojen luomisen. Fyysiset verkkoportit ovat esimerkiksi yleisesti tietokoneista löytyvä RJ-45-portti tai WLAN-kortti. Kaikki tietoverkon data kulkee verkkoporttien läpi, minkä takia jokaisella tietokoneella tai suorituskontilla täytyy olla ainakin yksi aktiivinen toiminnallinen verkkoportti, joka on määritetty siihen kytkettyyn verkkoon oikein.

Versiohaara (Git)

Lähdekoodista voidaan luoda uusia haaroja, joihin voidaan tehdä uusia solmuja tai muodostaa uusia haaroja. Vastaavasti haaroja voidaan yhdistää (merge) toinen toisiinsa, sekä haaraan, mistä kyseinen haara on peräisin. Tämä malli mahdollistaa helposti esimerkiksi eri ominaisuuksien tai bugikorjauksien jaottelun omiin haaroihinsa, kuin myös testausympäristön ja tuotantoympäristön lähdekoodin helpon erittelyn.

Yksikkötestaus

Yksikkötestaus, englanniksi ”unit testing” on ohjelmistonkehityksessä käytettävä tapa testata ohjelmiston toimintaa. Yksikkötestauksessa ohjelmiston yksittäiset osa-alueet testataan erikseen, jotta varmistetaan, että kyseinen osa-alue toimii suunnitellulla tavalla ilman, että ulkopuoliset komponentit vaikuttavat testin kulkuun tai tuloksiin. Yksikkötestit pyrkivät testaamaan komponenttia mahdollisimman monella eri tavalla, jotta mahdolliset poikkeustilanteetkin ovat huomioitu.

1 Johdanto

Nykyiset verkkopalvelut käyttävät komplekseja verkkojärjestelmiä, jotka kehittyvät jatkuvasti. Myös verkkopalveluiden saavutettavuus on jatkuvasti korostuvassa asemassa, minkä takia tietoverkkojen läpi kulkevan datan määrä kasvaa jatkuvasti. Verkkopalvelulle toiminnan varmuus, luotettavuus ja turvallisuus on äärimmäisen tärkeää saavutettavuuden maksimoimiseksi, joten palvelun ylläpitäjät haluavat pitää huolen siitä, että palvelun toiminta olisi valvottua sekä mahdollisimman katkeamatonta. Yksi olennainen osa tätä on palvelun verkkoliikenteen hallinta ja valvonta, jotta mahdolliset poikkeamat havaittaisiin mahdollisimman pian tai että mahdolliset ongelmatilanteet voitaisiin ratkaista jopa automaattisesti.

Verkkoliikenteen valvonta tapahtuu eri protokollatasoilla. Valvonnalla voidaan protokollan mukaan pitää kirjaa esimerkiksi siitä, mitkä laitteet ovat yhteyksissä keskenään, sekä tietyllä tasolla myös seurata, mitä dataa laitteiden välillä kulkee ja tämän avulla esimerkiksi havaita mahdolliset pahat tahot. (Cecil 2023.) Automatisoidun valvonnan avulla voidaan toteuttaa automaattista verkon hallintaa, esimerkiksi estää palvelunestohyökkäyksiä tai estää tiettyjen IP-osoitteiden pääsy palomuurin läpi sisäverkkoon, mikäli IP-osoitteesta tuleva liikenne havaitaan epätavalliseksi ja täten mahdollisesti pahatahtoiseksi.

Tämä opinnäytetyö keskittyy pitkälti verkkoliikenteeseen ja siinä ennen kaikkea liikenteen eri kerroksiin. Tätä tietopohjaa käytetään pitkälti työn toteutusvaiheessa, jossa varsinainen ohjelmisto ohjelmoidaan. Ohjelmistossa hyödynnetään tietoa verkkoliikenteen rakenteesta, jotta sen purkaminen olisi mahdollisimman monipuolista ja virhesietoista. Ohjelmiston kehitysvaiheesta on tehty muistiinpanoja ja koko prosessi on osana tätä raporttia. Työn lopussa esitellään yksinkertainen demo, jossa ohjelmiston toimintaa ja mahdollisuuksia havainnollistetaan, mutta itse varsinaisiin kompleksimpeihin käyttökohteisiin ja niiden havainnollistamiseen työssä ei keskitytä.

2 Lähtökohdat

Tämä opinnäytetyö on toteutettu Nolwenture Ltd:lle ja työn tarkoituksena on perehtyä aiheen kannalta olennaisiin aineistoihin ja tietolähteisiin, sekä toteuttaa ohjelmisto, joka tarjoaa verkkoliikenteen analysoinnin ja eri verkkokerrosten tietueiden purkamisen yksinkertaisena kehitysrajapintana, jonka avulla ohjelmiston päälle on helppo lisätä omaa logiikkaa käyttökohteen mukaan, missä liikenteestä saatua tietoa voidaan hyödyntää. Opinnäytetyössä toteutetulle ohjelmistolle on mahdollisia käyttökohteita toimeksiantajalla, mutta ohjelmistosta tulee avoimen lähdekoodin paketti, sekä Docker-levykuva, joten kuka tahansa voi hyödyntää ja jatkokehittää ohjelmistoa omiin käyttökohteisiinsa sopivaksi.

Toteutustekniikat ovat rajattu Docker-kontitusohjelman sekä PCAP-rajapinnan käyttöön. Muut asiat, kuten ohjelmointikieli, ovat omavalintaisia. Ohjelmisto toteutetaan TypeScript-ohjelmointikieltä käyttäen, koska kielelle löytyvät valmiit linkitykset (bindings) PCAP:lle ja ohjelmointikielistä minulla on eniten kokemusta TypeScript:llä. Toisena vaihtoehtona pidin myös Rust-ohjelmointikieltä. Koska kyseinen ohjelmointikieli on ”matalatasoista”, eli lähdekoodi on rakenteellisesti ns. lähellä sitä ajavaa laitteistoa, olisi ohjelmiston toteuttaminen paljon työläämpää, joskin paremmin optimoitua.

Opinnäytetyön tiedonkeruussa keskitytään pääasiassa PCAP-rajapintaan, OSI-malliin, mallin eri kerrosten standardeihin sekä Docker-kontitusohjelmistoon. Kaikki nämä ovat opinnäytetyön kannalta olennaisia, koska opinnäytetyössä toteutetaan PCAP-rajapintaa hyödyntävä ohjelmisto, joka kykenee lukemaan eri OSI-mallin kerrosten tietueita. Docker on myös keskeisessä asemassa, koska ohjelman lopullinen versio tulee pyörimään Docker-kontissa, mikä mahdollistaa ohjelman helpon siirrettävyyden.

2.1 Oman osaamisen lähtökohdat

Tietoverkkoliikenteen rakenne bittitasolla on minulle uusi konsepti. Tiesin ennen työn aloittamista karkeasti verkkoliikenteen eri tasot, mutta niiden merkitykset ja rakenteet ovat olleet pitkälti uutta asiaa. Pääasiassa ennakkotietämys on keskittynyt sovelluskerrokseen HTTP-protokollalla, sekä verkkoprotokollakerrokseen TCP- ja UDP-protokollien kautta. Näistäkään kerroksista varsinaiseen bittitasoon en ole ennen työtä perehtynyt, joten paljon opiskelua näihin liittyen tuli tehdä heti alussa.

Varsinaisen ohjelmoinnin puolella toteutusmenetelmiin löytyi kohtalaisen vahva ennakko-osaaminen. Typescript-ohjelmointikieli on ennalta tuttu, koska sitä on tarvittu niin työelämässä, kuin myös koulun sekä vapaa-ajan projekteissa. Git-versionhallinta on myös ennalta tuttu samoissa ympäristöissä. Testaukseen pohjautuva ohjelmistonkehitys on myös konsepti, joka on tullut varsin tutuksi työelämässä, mutta muualla en konseptia ole aiemmin käyttänyt. Docker-kontitusohjelmisto ja Docker-kontit ovat myös pääasiassa tulleet tutuiksi työelämän kautta, mutta Dockerin tarjoama tietoverkkorakenne, kuten virtuaaliset aliverkkokerrokset ja niiden liittäminen pääjärjestelmän verkkoliikennöintiin, puolestaan on pitkälti uutta asiaa.

2.2 Tietolähteet ja konseptit

Tiedonkeruulla pyritään syventämään omaa osaamista aiheesta sekä löytämään tieteellisiä artikkeleita ja tutkimuksia aiheeseen liittyen. Näitä tutkimuksia sekä niistä tehtyjä havaintoja ja tuloksia voidaan hyödyntää omassa toiminnassa. Tiedonkeruu keskitetään pääasiassa eri verkkokerroksiin, PCAP- kirjaston toiminnallisuuteen ja sen eri käyttömahdollisuuksiin sekä siihen, kuinka sitä on jo käytetty muissa tutkimuksissa ja myös opinnäytetöissä.

Toiminnallisessa opinnäytetyössä ratkaistaan jokin käytännön ongelma, tässä tapauksessa toteutetaan halutunlainen ohjelmisto ja toteutusprosessi dokumentoidaan. Hyvin toteutettu tiedonkeruu auttaa ohjelmistoratkaisun suunnittelussa ja toteutuksessa, sillä se takaa vankan pohjan työn tekemiselle, sekä selkeän käsityksen tiedonkeruun kohteista. Vankalla tietopohjalla varmistetaan, että ohjelmistossa toteutettavat toiminnot vastaavat odotuksia ja että yleiskäsitys ohjelmiston rakenteesta ja sisäisestä toimintamallista on selkeä.

3 Tiedonkeruu

3.1 PCAP

Tiedonkeruussa kerätään lähtökohtaisesti ajantasaista käytännön tietoa esimerkiksi olemassa olevien tieteellisten tutkimusten ja projektien, kuin myös kyselyiden, havaintojen ja haastatteluiden kautta. Tietoa pyritään myös keräämään monesta eri näkökulmasta, jotta aiheesta saadaan mahdollisimman kattava yleiskuva. Laaja kuva aiheesta auttaa myös mahdollisien vaihtoehtoisten ratkaisujen ja toteutustapojen löytämisessä. Koska kyseessä on tieteellisesti luotettava työ, täytyy tietolähteiden olla luotettavia ja lähteiden merkittynä.

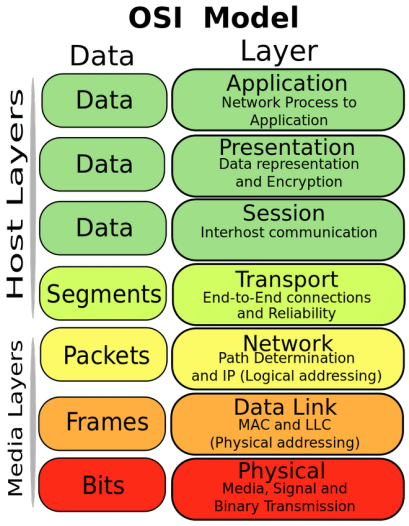
Packet Capture, eli lyhennettynä PCAP, on tekniikka, jonka avulla voidaan kaapata verkkoliikennettä reaaliajassa. Suomenkielinen nimitys tekniikalle on pakettianalysointi. PCAP itsessään ei ole varsinaisen käytettävän API:n nimi, vaan enemmänkin geneerinen nimitys toiminnallisuuden tarjoavalle API:lle. PCAP hyödyntää montaa eri työkalua, joiden avulla se kykenee tarjoamaan niin verkkoliikenteen datapakettien nappaamisen kuin myös pakettianalyysin. Tämän työn kannalta olennaisempi näistä kahdesta on datapakettien nappaus. Yleensä datapakettien nappaajat ovat virtuaalisia ohjelmistoja, mutta fyysisiäkin pakettikaappaajia on. (Dellinger & Aditham 2023.)

Lähtökohtaisesti PCAP luo tiedoston, jonne se kykenee tallentamaan kaiken verkkoliikenteen myöhempää analyysiä varten. Tässä työssä verkkoliikenne puolestaan suoratoistetaan prosessoitavaksi. Varsinaiset PCAP:n ohjelmistorajapinnat ovat Unix-pohjaisilla järjestelmillä libpcap ja Windowsilla Npcap, joka on seuraaja vanhentuneelle WinPcap- ohjelmistorajapinnalle (Ruoho 2019). Pakettianalyysiohjelmia löytyy niin ilmaisia kuin myös maksullisia, esimerkiksi Wireshark ja tcpdump, jotka ovat molemmat ilmaisohjelmistoja, joista toinen tarjoaa graafisen käyttöliittymän ja toista ajetaan terminaalista käsin. Pakettianalyysistä löytyy olemassa olevia tutkimuksia, joissa hyödynnetään joko pakettianalyysia hyödyntävää ohjelmaa tai tehdään jotakin suoraan pakettianalyysin ohjelmointirajapinnalla.

3.2 OSI-malli ja TCP/IP-protokollat

Open Systems Interconnection, lyhennettynä OSI-malli on kehitetty, jotta verkkojärjestelmien eri kerrokset olisi helppoa erotella toisistaan, sekä luetella eri kerroksien tehtävät (ISO/IEC 1994.). OSI-malli oli ensimmäinen mallistandardi verkkojärjestelmille ja nykypäivän internet pohjautuukin TCP/IP-malliin OSI:n sijaan. Vaikka OSI-malli onkin jo vanhentunut malli, on se edelleen laajasti käytössä tänäkin päivänä. (Impreva 2023.)

OSI-malli koostuu seitsemästä eri kerroksesta (kuvio 1). Kukin kerroksista toteuttaa oman protokollansa mukaisen toiminnallisuuden ja välittää kapseloidun datan seuraavalle kerrokselle. OSI-malli on standardi verkkopinon rakenteelle ja toimii pohjana pinon sisältämille standardeille. Kyseessä on siis standardi tuleville standardeille. (ISO/IEC 1994.) Yksinkertaisesti ajateltuna dataa lähettäessä tieto etenee mallia ylhäältä alaspäin ja dataa vastaanottaessa alhaalta ylöspäin (Tuomisto 2022.).



Kuvio 1: OSI-malli ja sen kerrokset (Lifewire. 2024.)

OSI-mallin alin kerros on fyysinen kerros, jonka päävastuu on lähettää data raakana bittivirtana sille määritellyllä bittinopeudella fyysistä tiedonsiirtovälinettä käyttäen, esimerkiksi verkkokaapelia tai valokuitua pitkin. Myös langaton verkko on osa fyysistä kerrosta, mutta datan lähettämisessä ja vastaanottamisessa käytetään radioita. (Shaw 2018.)

Toinen kerros on datalinkkikerros, jonka päätehtävä on vastata yhteyksien muodostamisesta fyysisillä kerroksilla. Tämä kerros myös vastaa sille annetun datan paloittelusta, synkronoinnista ja virheenkorjuusta. Datalinkkikerros hyödyntää verkkolaitteiden MAC-osoitteita laitteiden tunnistamisessa. (Shaw 2018.) OSI-mallin datalinkkikerrosta ei pidä sekoittaa nykyaikaisemman TCP/IP-mallin linkkikerroksen kanssa. Tämän kerroksen tiedonsiirto tapahtuu kahden laitteen välillä, konseptia laitteiden ketjuttamisesta ei vielä ole (Tuomisto 2022.). Datalinkkikerros toimii pitkälti yhdessä verkkokerroksen kanssa datansiirrossa ja esimerkiksi datalinkkikerroksen laiteosoitteiden selvittämisessä ja yhdistämisessä niiden vastaaviin IP-osoitteisiin käytetään ARP-protokollaa, joka toimii käyttäen sekä verkkokerroksen, että datalinkkikerroksen tietoja. ARP-paketti voi esimerkiksi sisältää lähde- ja kohdelaitteiden IP-osoitteet ja ARP-pyyntö sisältää lähteen MAC-osoitteen ja vastaus puolestaan sisältää kohteen MAC-osoitteen.

(Arkko & Pignataro 2009.) Datalinkkikerroksessa voi olla myös lisäksi esimerkiksi standardin 802.1Q mukainen VLAN-tunnus (IEEE 2003.).

Kolmas kerros on verkkokerros. Tämän kerroksen päätehtävä on pilkkoa data verkkopaketeiksi ja määrittää paketeille reitti, jota pitkin ne pääsevät helpoiten kohteeseensa. Tämä kerros tarjoaa siis mahdollisuuden tehdä monen verkkolaitteen hypyn paketinvälitystä. Tunnetuin protokolla tällä kerroksella on IP (Internet Protocol). (ISO/IEC 1994.) Tässä kerroksessa ei enää tarvitse välittää siitä, miten itse varsinainen tiedonsiirto tapahtuu (Tuomisto 2022.).

Neljäs kerros vastaa tiedon osioinnista ja vastaanottaessa uudelleenkasauksesta siten, että data on käyttökelpoista ja luettavaa ylemmällä tasolla. Vastaavasti taso vastaa myös datassa havaittavista virheistä, jos data on vioittunutta, voi kerros pyytää lähettäjää lähettämään kyseisen datan osan uudelleen. Tunnetuimmat protokollat tällä kerroksella ovat TCP ja UDP. (Shaw 2018.) TCP on yleisin protokolla, koska se vastaa ehyen tiedonsiirron ja protokolla on hyvin vanha, vaikkakin sitä on päivitetty paljon. Protokollan tiivis sidonnaisuus IP-verkkoihin on havaittavissa jo pelkästään verkkostandardin nimestä (TCP/IP). TCP takaa, että paketit vastaanotetaan samassa järjestyksessä kuin missä ne on lähetetty ja että kaikki paketit tulevat onnistuneesti perille. Yksinkertaistettuna tämä tapahtuu lähettämällä datapaketti ja paketin tiiviste kohdelaitteelle, joka vertaa paketista laskettua tiivistettä mukana tulleeseen tiivisteeseen. Jos tiivisteet ovat yhdenmukaiset, on paketti ehyt ja kohdelaite lähettää ACK-paketin lähettäjälle ilmoittaakseen, että paketti on tullut perille ja samalla olevansa valmis vastaanottamaan seuraavan paketin. Koko tämän ajan lähettäjä odottaa ja mikäli ACK-pyyntöä ei kuulu tiettyyn aikaan mennessä, paketti yritetään lähettää uudelleen niin pitkään, kunnes pakettia vastaava ACK-pyyntö saapuu takaisin. (Eddy 2022.)

Viides kerros vastaa verkkoistunnoista. Se takaa, että istunnot avataan ja ovat käytettävissä tiedonsiirtoon, kun niitä tarvitaan sekä sulkee istunnon, kun sitä ei enää tarvita. Esimerkiksi ”socketit” toimivat istuntokerroksella. Jokaisella istunnolla on oma datasocket, joka pidetään auki niin pitkään kuin sen on tarve. (ITU

1996.) Vastaavasti verkkopalvelin, joka kuuntelee verkkoliikennettä, odottaa istuntokerroksella uusien istuntojen luontia ja avaamista.

Kuudes kerros on esityskerros. Sen tehtävä on vastata siitä, miten tieto esitetään sovelluskerrokselle. Vastaavasti tietoa alemmille kerroksille välittäessä sen tehtävänä on määrittää datan koodekki ja salaustekniikka (kryptays), kuin myös datan pakkaus. Salaustekniikka voi olla esimerkiksi jokin TLS-salauksen eri versioista. Tämä kerros ottaa vastaan kaiken lähetettävän datan, jota tulee sovelluskerrokselta ja välittää sen alaspäin. (Shaw 2018.)

OSI-mallin ylin kerros, seitsemäs kerros on sovelluskerros. Tämä kerros sisältää ohjelmistot ja niiden omat kommunikaatiostandardit, kuten selaimet, sähköposti ja nimenselvennyspalvelimet. Näille omia standardeja ovat esimerkiksi HTTP, FTP, POP3, SMTP ja DNS. (Shaw 2018.) Tunnetuin näistä on HTTP, joka on selkotekstillä koodattua ja toimii pyyntö- vastausperiaatteella. HTTP on myös tilaton sekä käyttöliittymäriippumaton, joka tekee siitä hyvin joustavan ja monipuolisen erilaisille laitteistoille ja käyttökohteille. (Fielding & Nottingham & Reschke 2022.)

3.3 Docker

Docker on yksi suurimmista tänä päivänä käytössä olevista kontitusohjelmista. Ohjelmistojen kontittamisen ideana on taata ohjelmien helppo suorittaminen suoritusympäristöstä riippumatta sekä rajata ohjelmiston pääsyä isäntäjärjestelmän resursseihin, kuten laitteistoon, massamuisteille sekä verkkoon. (Babak & Harrison & Mohammed 2017.) Kontitus on pitkälti korvannut virtuaalitietokoneiden roolia palvelinympäristöissä, koska kummatkin tavoittelevat samaa asiaa, mutta kontitus toteuttaa sen pienemmällä resurssimäärällä. Lisäksi Docker-kontit käyttävät isäntäjärjestelmän ydintä (eng. kernel), mikä tuo paremman suorituskyvyn kontissa ajettaville ohjelmistoille, sekä poistaa tarpeen kokonaisen vieraskäyttöjärjestelmän olemassaololta pelkästään kontitetun ohjelmiston ajamista varten. (Docker 2024.)

Docker-kontti sisältää yksinkertaisen ajoympäristön, johon ajetaan sille annettu levykuva, joka sisältää resurssit ohjelmiston suorittamiseen, kuten ohjelmiston tarvitsemat jaetut resurssit sekä ajavan ohjelman (Babak & Harrison & Mohammed 2017). Node-pohjaisilla ohjelmistoilla tämä tarkoittaa käytännössä kevyttä linux-jakelua ja node.js-suoritusohjelmaa. Näiden päällä voidaan ajaa toteutettua JavaScript-ohjelmaa (Juell 2022.) Ohjelman tarpeiden mukaan saatetaan tarvita laajennettua pääsyä isäntäjärjestelmän resursseihin tai ajoympäristö tarvitsee jotakin lisäosia, sekä portinavausta kontille, jotta kontti voi jakaa sisältöä, esimerkiksi verkkopalvelimen. (Babak & Harrison & Mohammed 2017.)

4 Toteutustekniikat

Saatujen tietojen pohjalta opinnäytetyötä on helppoa lähteä työstämään. Kerätyn tiedon avulla on saatu käsitys siitä, mitä rajapintaa ja ohjelmia käytetään ja mitä ominaisuuksia ne tarjoavat. Tämän työn kannalta olennaisin toiminnallisuus on itse verkkoliikenteen kaappaus kloonaamalla datavirta siten, että se ei vaikuta alkuperäiseen liikenteeseen. Lisäksi kyseiseen toiminnallisuuteen liittyen on olemassa paljon dokumentaatiota sekä myös toiminnallisuutta hyödyntäviä valmiita ohjelmia.

Jotta verkkoliikenteestä saadaan mitään dataa ymmärrettävään muotoon, tulee ymmärtää sen rakenne. Tässä OSI-mallin ymmärtäminen on suuri apu, koska vaikka malli onkin vuonna 2023 vanhanaikainen, on perusrakenne vielä nykyisinkin käytännössä sama. Pääasiassa malli on internet-standardeissa pelkistetty vähempikerroksiseksi, mutta on säilynyt protokollatasoon asti käytännössä muuttumattomana. Tämän työn kannalta olennaiset verkkokerrokset ovat pysyneet samana.

Koska tässä työssä Dockeria käytetään verkkomonitorointityökalun kontittamiseen, tulee Dockerin mahdollistaa tarvittava pääsy isäntäjärjestelmän verkkoportteihin, niin fyysisiin kuin myös virtuaalisiin. Docker mahdollistaa tämän onneksi network-parametrin avulla. (Docker 2023.) Koska Docker voi ottaa vaikka koko isäntäjärjestelmän verkkoadapterit haltuun, voidaan sille antaa pääsy tarvittaviin verkkopäätteisiin, jotta dataa voidaan lukea.

TypeScript-ohjelmointikielelle löytyy tarvittavat kirjastot, jotka tarjoavat pääsyn PCAP:n ohjelmistonkehitysrajapintaan ja sen tarjoamiin metodeihin. Tällä hetkellä suurin julkisesti saatavilla oleva kirjasto on node_pcap, joka on avoimen lähdekoodin kirjasto sekä löydettävissä npm-palvelusta nimellä ”pcap”.

Ohjelmiston toteuttaminen näillä resursseilla on siis mahdollista ja suureen osaan resursseista löytyy myös kattavasti referenssimateriaalia internetistä. Tähän kirjallisuuskatsaukseen on koostettu niistä toteutuksen kannalta olennaisimmat tiedot. Varsinaisiin ohjelmointiin liittyviin yksityiskohtiin mennään tarkemmin vasta toteutusvaiheessa.

5 Käytettävät työkalut ja menetelmät

Ohjelmistonkehityksessä käytetään paljon erinäisiä työkaluja ja ohjelmia kehittämisen tukena ja virtaviivaistamisena. Välttämättömät työkalut ovat lähes poikkeuksetta lähdekoodin muokkaamiseen soveltuva tekstieditori, sekä ohjelmointikielelle yhteensopiva koodinkääntäjä. Tässä projektissa on käytössä myös lisäksi Docker-kontitusohjelmisto, versionhallintatyökalut sekä koodieditorin tarjoamat lisätyökalut.

Ohjelmointi toteutetaan testauspohjaisella kehitysmallilla, eli ohjelman kriittiset osa-alueet ovat huolella testattuja yksikkötestien avulla. Tässä projektissa testaukseen käytettiin valmista Jest-nimistä kirjastoa, joka löytyy esimerkiksi NPM-pakettikokoelmasta. Koodieditorina ja IDE:nä käytössä on Visual Studio Code.

Koodin kääntämisestä, sekä reaaliaikaisesta analysoinnista vastaa Visual Studio Codeen lisäosana asennettava Typescript-laajennus.

Versionhallinnasta ja koodin pilvisynkronoinnista vastaa GitHub-palvelu, joka on myös integroitu Visual Studio Coden käyttöliittymään. Koska projekti on toteutettu yksin, eikä rakenteellisesti ole varsin kompleksi, ei tässä projektissa ole käytetty versiohaaroja vaan pelkästään gitin solmuja. Versionhallinnasta on hyötyä myös siinä, että ohjelmistonkehitystä on tehty käyttäen kahta erillistä tietokonetta: Pöytätietokonetta sekä kannettavaa tietokonetta ja versionhallinta mahdollistaa helposti koodin kloonauksen ja ajan tasalla pitämisen molemmilla järjestelmillä.

Ohjelmisto on toteutettu siten, että sitä voidaan helposti ajaa sellaisenaan millä vaan järjestelmällä, jossa on asennettuna PCAP-API ja NodeJS, kuin myös Docker-kontissa. Ohjelmiston modulaarinen rakenne mahdollistaa helpon käyttöönoton, sekä ohjelmiston tarjoama tapahtumapohjainen API antaa hyvän pohjan tehokkaalle ja asynkroniselle jatkokehitykselle. Koska kyseessä on verkkodataa reaaliaikaisesti käsittelevä ohjelmisto, täytyy ohjelmiston suorituksen olla asynkronista, jotta ohjelma ei ylikuormitu datan prosessointia odottaessa.

6 Ohjelmiston toteutus

6.1 Kehityksen aloitus

Ohjelmiston kehittäminen alkoi marraskuun 2023 alussa. Projektille luotiin Git-repositorio, ja projektin työtunneille ja muistiinpanoille tehtiin tekstitiedosto. Varsinainen ohjelmistonkehitys alkoi tästä etenemään hiljalleen työtunteja sekä kehityksen aikana tehtyjä havaintoja kirjaten. Versionhallintaan lähdekoodia päivitettiin sitä mukaan, kun siihen tehtiin jotakin merkittävää tai oli tiedossa, että kehitys jatkuu seuraavalla kerran toisella alustalla.

Ensimmäinen vaihe ohjelmistonkehityksessä oli tutustua käytettäviin rajapintoihin. Kehitysympäristö Visual Studio Code oli entuudestaan tuttu useamman vuoden käyttökokemuksella. Lisäksi Docker, sekä Docker Desktop ovat tulleet töiden kautta tutuksi ja osaksi normaalia kehitysprosessia. Sen sijaan PCAP-rajapinta sekä sen tarjoama verkkodata ovat molemmat vieraita. Verkkodatan rakenne konseptitasolla tuli tutuksi tiedonkeruuvaiheessa, joten nyt oli aika perehtyä datan rakenteeseen bittitasolla.

Datan rakenteen tulkitseminen alkoi OSI-mallin alimmalta PCAP:lle saatavilla olevalta kerrokselta, eli linkkikerrokselta. Linkkikerroksessa kulkevat datapaketit voidaan karkeasti jakaa bittitasolla kahteen osioon: otsakkeisiin ja varsinaiseen dataan. Otsakkeet sisältävät datan, mitä tästä kerroksesta halutaan purkaa, kuten lähde MAC-osoitteen, kohde MAC-osoitteen, mahdollisen VLAN-tunnisteen ja sen tietueet, kuten ID:n ja prioriteetin, sekä EtherTypen.

Aivan ensimmäiseksi asensin tietokoneelleni libpcap-kirjaston version 1.10.4-1, joka oli viimeisin Arch Linuxille saatavilla oleva versio. Kirjasto on välttämätön järjestelmässä, jossa ohjelmistoa ajetaan. Seuraavaksi lisäsin pcap npm-moduulin osaksi ohjelmiston riippuvuuksia, koska moduuli tarjoaa linkityksen pääjärjestelmän libpcap-kirjastoon ja täten pääsyn raakaan datavirtaan, mitä pcap näkee.

Aivan alkuun tein yksinkertaisen testiohjelman, joka avaa PCAP-datavirran ja kirjaston tarjoaman tapahtumapyynnön kautta antaa raa’an bittijonon jokaisesta ethernet-kehyksestä, joka sisältää verkkodatan bitteinä. Lisäksi testasin kirjaston sisäistä paketinpurkua, joka osoittautui melko tehokkaaksi. Kirjasto purki ethernet-kehyksen, sekä sen sisältämän IP-paketin perustiedot, mutta oman oppimisen ja ohjelmiston modulaarisuuden kannalta tätä paketinpurkutoiminnallisuutta ei tässä opinnäytetyössä käytetä muuhun, kuin kirjaston tarjoamaan toiseen toiminnallisuuteen, TCP-trackeriin. TCP-tracker tarjoaa kaksi tapahtumapohjaista takaisinkutsufunktiota, joista toinen kutsutaan uuden TCP-yhteyden avauksen yhteydessä ja toinen sen sulkeutuessa. Funktion takaisinkutsun mu-

kana tuleva data tarjoaa kattavan määrän tietoa TCP-yhteydestä, mutta olennaisinta tällä ominaisuudella on tarjota mahdollisuus pitää kirjaa aktiivisten TCP-yhteyksien määrästä, niiden datavirran suuruudesta ja mihin IP-osoitteisiin yhteydet johtavat.

Ennen kuin aloin varsinaisesti kehittämään ohjelmistoa, toteutin huolellisen suunnittelun. Etenkin sellaisissa ohjelmistoissa, joita on tarkoitus jatkokehittää ja skaalata myöhemmin, on tärkeää, että ohjelmiston rakenne on toteutettu siten, että se mahdollistaa sen. Tämän takia otin modulaarisuuden ja tapahtumapohjaisen suorituksen ohjelmiston rakenteen tavoitteiksi. Ideaalisinta olisi, että datan purku tapahtuisi kerroksittain ja jokainen kerros toimisi itsenäisesti riippumatta siitä, mistä sen saama data on peräisin. Tämä on mahdollista, koska OSI-mallin kerrokset ovat lähes täysin riippumattomia toisistaan.

Tapahtumakutsu-pohjainen tapahtumanhallinta puolestaan mahdollistaa samanaikaisen prosessoinnin usealle paketille, sekä minimoi kuormituksen. Syynä tähän on se, että ohjelma ei kuluta turhaan käyttöjärjestelmän tehtäväaikataulua esimerkiksi nukuttamalla ydintä odottaessaan uutta pakettia vaan osaa ilmoittaa olevansa jouten, mikäli prosessoitavaa dataa ei ole. Tämän seurauksena käyttöjärjestelmä osaa antaa ns. ylimääräisen suoritusajan muille prosesseille. Etenkin tuotantoympäristöissä, jossa ohjelmistojen tehokkuudella on merkitys ylläpitokuluissa ja turha kuorma halutaan minimoida, tämä on tärkeää.

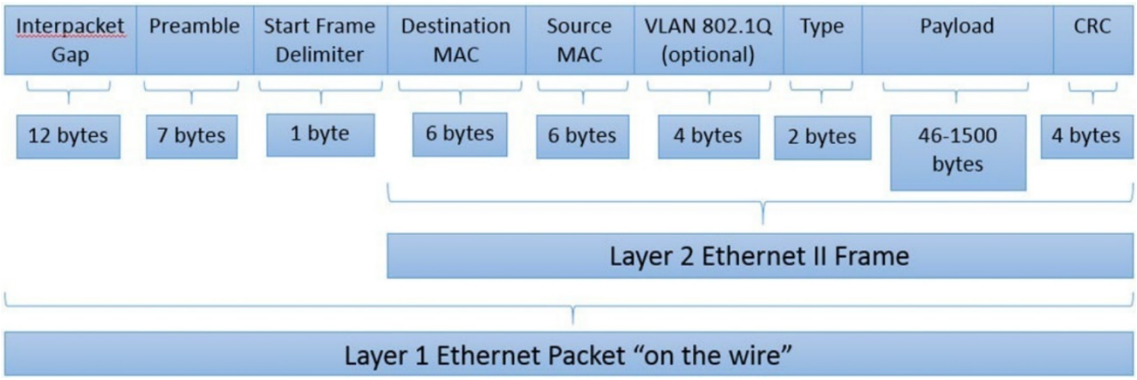
Luotuani tämän idean ohjelmiston rakenteesta ja toimintamallista, aloin suunnittelemaan, missä järjestyksessä lisään toiminnallisuutta ohjelmaan, toisin sanoen, mikä on ensimmäinen prioriteetti. Tärkeintähän ohjelmistolle on ylipäätään saada olennaista tietoa ulos PCAP-datasta, joten se vaikuttaisi parhaalta kohdalta aloittaa. Datan paketointi API:ksi on myös tärkeää, mutta ei yhtä suurella prioriteetilla ja lisäksi se olisi hankala suunnitella ja toteuttaa etukäteen, kun itse paketoitavaa dataa ei vielä olisi.

Itse datan purkaminen puolestaan kannattaa aloittaa OSI-mallin pohjalta. Syynä tähän on OSI-mallin kapseloiva toimintaperiaate, eli alempi kerros kapseloi

ylemmän kerroksen datan payloadinaan. Täten kun alemman kerroksen datat ovat purettu auki ja eroteltu, on alemman kerroksen payload helppo ottaa prosessoitavaksi ylemmän kerroksen datan purkajaan. Tämä käy hyvin yhteen modulaarisen rakennesuunnitelman kanssa. Tässä vaiheessa suunnitelma oli selvä: mitä, miten ja missä järjestyksessä. Kehitysprosessi voi nyt alkaa ja ensimmäisenä kehityskohteena on datan purku alimmasta saatavilla olevasta kerroksesta, tässä tapauksessa linkkikerroksesta, jonka data on saatavilla joko ethernet-paketteina tai-kehyksinä. Libpcap tarjoaa omassa kehitysympäristössäni tämän kerroksen datan ethernet-kehyksinä niin Docker-ympäristössä, kuin myös isäntäkoneella suorittaessa, joten datan purkaminen on toteutettu niille.

6.2 Ethernet-kehyksen purkaminen

Perehdyttyäni ethernet-kehyksen bittirakenteeseen, aloitin ohjelmiston varsinaisen kehittämisen. Loin ohjelmiston koodiin uuden tiedoston, joka tulisi sisältämään funktionaalisen komponentin puhtaasti ethernet-kehyksen datan purkamiseen. Datan purku keskittyy pääasiassa otsakkeiden dekoodaamiseen. Suoraan PCAP:sta saadun ethernet-kehyksen data on raakaa binääridataa siinä muodossa, jossa se tulee kulkemaan bittijonona lopulta OSI-mallin alimman kerroksen, eli fyysisen kerroksen sisällä. Fyysisen kerroksen yhteys voi olla niin Ethernet-yhteys, parikaapeli, tai jopa langaton yhteys, kuten WiFi. Kuten voimme päätellä, data ei ole juurikaan sellaisenaan luettavissa tai käytettävissä, vaan datasta tulee purkaa irti varsinaiset ohjelmistolle olennaiset tiedot. Tämän vuoksi ethernet-kehyksen raa’an bittirakenteen ymmärtäminen on tärkeää ja myös tarkoin määritelty IEEE 8802-3 -standardissa. Standardi määrittelee niin otsakkeiden rakenteen, järjestyksen, kuin myös pituuden tavuina (kuvio 2).



Kuvio 2: Ethernet-kehyksen rakenne bittitasolla (Dell Technologies. 2024.)

Aivan alkuun aloin toteuttamaan funktiota, joka saa raakamuotoisen ethernet-kehyksen syöteparametrina ja lukee siitä ulos kehyksen lähde ja kohde MAC-osoitteet. Koska tässä tapauksessa kyseessä ovat Ethernet II -tyypin ethernet-kehykset, ovat osoitteet heti kehyksen ensimmäiset otsakkeet. Standardin mukaan tason 1 ethernet-verkkodatassa kyseessä olisivat ethernet-kehysten sijasta kokonaiset ethernet-paketit ja näissä paketeissa MAC-osoitteet alkaisivat vasta tavuindeksistä 8. Lähdeosoite ensin ja heti seuraavaksi kohde, molemmat pituudeltaan kuusi oktettia, eli 6 * 8 = 48 bittiä. Tietojen purkaminen osoittautui lopulta melko yksinkertaiseksi: Leikataan raakadatan bittijonosta kuuden oktetin kokoinen pala oikeasta aloittamiskohdasta, esimerkkinä kohdeosoitteessa tavuindeksi nollasta ja luetaan data string-tyypiksi heksadesimaalimuodon konversiota käyttäen. Lopputuloksena on string-muuttuja, joka sisältää MAC-osoitteen yhtenäisenä merkkijonona.

Kun tiedot ovat purettu ulos raa’asta bittijonosta, luodaan niistä uusi json-objekti ja objekti palautetaan takaisin tuloksena metodin kutsuneelle isäntäfunktiolle. Tässä vaiheessa isäntäfunktio oli yksinkertainen, suoraan index.ts-tiedostoon tehty takaisinkutsufunktio (callback function) joka ajetaan aina, kun PCAP-kirjaston ”raw”-tapahtuma kutsutaan. Tämä tapahtuma puolestaan tarjoaa takaisinkutsufunktiolleen dataparametrin, joka sisältää itse ethernet-kehyksen raa’an binääridatan, minkä libpcap-API on suoraan kaapannut sille määritellystä verkkoportista. Takaisinkutsufunktio puolestaan kutsuu ethernet-kehyksen purku-

funktion sekä antaa sille binääridatan syöteparametrina ja lopulta asettaa tuloksena tulleen objektin uuteen muuttujaan. Tämän jälkeen funktio tulostaa muuttujasta sen sisältämät puretut tiedot, tässä tapauksessa aiemmin määritellyt MAC-osoitteet. Yksikkötestejä ja varsinaista ohjelmiston lopullista rakennetta ei vielä tässä vaiheessa otettu huomioon, mutta modulaarisuus ja tapahtumapohjainen järjestelmä ovat heti alusta asti pääroolissa ohjelmaa toteuttaessa. Yhteensä tämän vaiheen toteuttamisessa kului aikaa noin 5 tuntia, sisältäen ajan, joka kului ethernet-kehyksen bittirakenteeseen perehtymiseen.

Tässä vaiheessa ethernet-kehysten purkajalla on luotu pohja, jonka päälle on helppo lähteä lisäämään muiden otsakkeiden datan purkamista. Edeten ethernet-kehyksen otsakkeiden mukaisessa järjestyksessä seuraavaksi otsakkeista tuli lukea ulos niin Ethertype, kuin myös mahdollinen standardin 802.1Q mukainen VLAN-tunnus (IEEE 2003.). Jos kehys olisi 802.3-formaatin mukainen, eli ns. ”raaka” kehys, EtherTypeä ei olisi vaan sen sijasta data ilmaisisi kehyksen koon. PCAP-API tarjoaa datan Ethernet II -muodossa, joten voimme olettaa datan olevan aina EtherType.

Tunnuksen purkaminen osoittautui kuitenkin huomattavasti MAC-osoitteiden purkua vaativammaksi tehtäväksi, koska toisin kuin MAC-osoitteet, jotka ovat aina määritelty ja samassa paikassa, VLAN-tunnuksen olemassaolo määritellään seuraavanlaisesti: Mikäli tunnus on määritelty, ovat ensimmäiset kaksi oktettia heksadesimaaliarvo 0x8100 ja seuraavat kaksi oktettia sisältävät varsinaisen TCI (Tag Control Information) -datan. Tämä data sisältää 3 bittiä pitkän kentän VLAN-verkon prioriteetille, 1 bitin boolean-arvon siitä, onko ethernet-kehys kelvollinen pudotettavaksi ruuhkautuessa vai ei, sekä 12-bittisen tietokentän VLAN-verkon heksadesimaaliselle tunnukselle (teoreettinen maksimimäärä on 4094 VLAN-verkkoa) (kuvio 3). Tämän datan jälkeen tulee Ethertype. Vastaavasti jos VLAN-tunnusta ei ole määritelty, onkin VLAN-tunnuksen olemassaolon indikoivan kahden oktetin asemassa EtherType. EtherType on kuitenkin aina vain kaksi oktettia pitkä. Kuten voimme nähdä, EtherType-otsakkeen sijainti ethernet-kehyksen otsakkeissa bittitasolla on 802.1Q-tunnuksen olemassaolosta riippuvainen.

802.1Q tag format

16 bits	3 bits	1 bit	12 bits
TPID	TCI		
	PCP	DEI	VID

Kuvio 3: 802.1Q-tunnuksen bittirakenne (Networkcorner. 2024.)

Ymmärrettyäni tämän konseptin, aloin suunnittelemaan, miten voisin mahdollisimman tehokkaasti ja mahdollisimman pienellä määrällä datan purkua lukea, onko tunnusta olemassa vai ei ja ajaa tunnuksen datan purkavan logiikan vain, mikäli on. Ideaalisinta olisi, jos ensimmäiset kaksi bittiä puretaan Ethertypenä ja tehdään vertausoperaatio heksadesimaaliarvoa 0x8100 kohti. Mikäli arvot täsmäävät, tiedämme, että otsake on VLAN-tunnus ja toteutamme tunnuksen datan purkamisen sekä määritämme EtherTypen uudelleen lukemalla seuraavat kaksi bittiä VLAN-tunnuksen jälkeen. Mikäli vertausoperaatio palauttaa epätoden tiedämme, että VLAN-tunnusta ei ole ja voimme jättää EtherTypen siihen arvoon, missä se jo valmiiksi on. Täten VLAN-tunnuksen puuttuessa koko operaatio koostuu yhdestä datanpurkuoperaatiosta ja vertausoperaatiosta. Vastaavasti jos VLAN-tunnus on määritelty, teemme samat operaatiot kuin tunnuksen puuttuessa, mutta lisäksi puramme tunnuksen datan ja dekoodaamme sen bittitasolla. Tämän jälkeen uudelleen määritämme EtherTypen oikeaan arvoonsa. Tämä ratkaisu vaikutti teoriatasolla tehokkaimmalta, koska siinä purkuoperaatioiden määrä pidetään minimaalisena.

Tässä vaiheessa ethernet-kehyksen otsakkeista on purettu irti kaikki data, joka on koettu tarpeelliseksi ja jäljellä onkin enää itse ethernet-kehyksen sisältämä kapseloitu data, eli payload. Koska koko kehyksen pituus ja otsakkeiden loppuosan sijainti ovat tiedossa, on payload helppo lukea ulos kehyksestä omaksi muuttujakseen. Tämän jälkeen puretut otsakkeet ja payload asetetaan uuteen json-objektiin, jolle on määritelty tyypitys (kuvio 4), eli tämän objektin sisältö on

aina samassa muodossa. Lopulta tämä muuttuja palautetaan return-funktiolla kontekstiin, josta metodi on suoritettu.

```
interface EthernetFrame {
  macDestination: string;
  macSource: string;
  vlanTag?: VlanTag;
  etherType: EtherType;
  payload?: Buffer;
}
```

Kuvio 4: Ethernet-kehyksen puretun datan rakenne (Kuvankaappaus ohjelmiston lähdekoodista. 2024.)

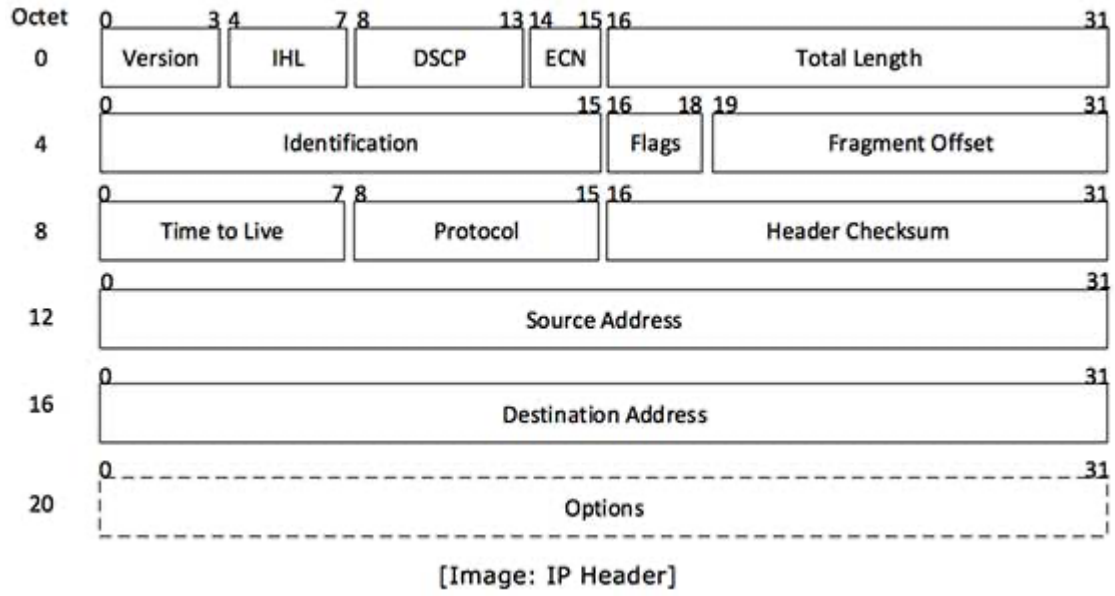
Ethernet-kehyksiä purkaessa eniten aikaa vei alkuun sisäistää idea kehyksen rakenteesta bittitasolla ja myös, kuinka kehyksen bittidataan pääsee kiinni mahdollisimman yksinkertaisesti ja tehokkaasti Typescriptillä. Etenkin hankalaksi osoittautui tilanteet, missä yksi tavu sisältää monta eri datakenttää, esimerkiksi VLAN-tunnuksen purkaminen. Lisäksi pohtimista ja suunnittelua vaati se, kuinka on järkevintä pitää kirjaa siitä, missä kohti pakettia purkaminen on menossa. Tämän olisi voinut toteuttaa kovakoodaamalla tavujen sijainnit kuhunkin purkuoperaatioon, joka olisikin ehkä ollut parempi ratkaisu, mutta kehittäessä ohjelmistoa tuntui loogiselta, että purkajalla on sisäinen ”offset”-muuttujansa, joka arvo vastaa nykyistä lukijan sijaintia ja tavujen lukuoperaatiot käyttävät tätä muuttujaa parametrinaan. Itse palautettavan datarakenteen suunnittelu ja toteutus puolestaan osoittautui nopeaksi ja yksinkertaiseksi. Datarakenne oli helppo muodostaa, kun tiesi, mitä dataa kehys sisältää ja siitä datasta valitsi vain kaiken olennaisen. Kaiken kaikkiaan ethernet-kehyksen purkajan ohjelmointiin kului aikaa noin 20 tuntia. Tässä vaiheessa kuitenkaan yksikkötestejä ei vielä ole toteutettu, vaan ne tulevat myöhemmin.

6.3 IP-verkkopaketin purkaminen

Ethernet-kehyksen purkamisesta saatu payload sisältää OSI-mallin seuraavan kerroksen datan. Tämä kerros on verkkokerros, eli tämä pitää sisällään varsi-

naisen verkkorakenteesta vastaavan datan ja sitä seuraavan kerroksen payloadin. Pääasiassa tässä kerroksessa kulkeva data on IP-verkkodataa, joko IPv4- tai IPv6-protokollalla. Tämän takia tässä ohjelmistossa verkkopakettien purku tukee vain IPv4- ja IPv6-protokollia, mutta jatkokehitys useammille protokollille on myös mahdollista.

Kuten ethernet-kehysten purkamisessa, IP-pakettien purkamisen toteuttaminen alkoi pakettien rakenteeseen tutustumisella. Ensin perehdyin yksinomaan IPv4-pakettien rakenteeseen. Samoin kuin ethernet-kehykset, Internet Protocol -paketit voidaan jakaa kahteen osaan: otsakkeisiin ja payloadiin. Näistä datan purkamisen kannalta olennaiset ovat itse otsakkeet (kuvio 5).



Kuvio 5: IPv4-paketin rakenne bittitasolla (Tutorialspoint. 2024.)

IPv4-paketissa otsakkeista ensimmäiset osat ovat internet paketin versio ja otsakkeiden pituuden ilmaisin. Nämä molemmat tietueet ovat saman oktetin sisällä neljän bitin kenttinä, joten datat sisältävä tavu pitää jälleen paloitella bittitasolla. Versio saadaan ulos siirtämällä tavun bittejä oikealle päin neljä pykälää ja suorittamalla bittikohtainen JA-operaatio suodattamaan ylimääräiset bitit pois. Tämän seurauksena jäljelle jäävät bitit kuvaavat tavun ensimmäisen neljän bitin arvoa, eli paketin versiota.

Otsakkeiden pituus noudattaa samaa kaavaa, mutta yksinkertaistettuna, koska bittejä ei tarvitse siirtää vaan niille voidaan suoraan toteuttaa bittikohtainen JA-operaatio, mikä suodattaa pois version ilmaisevat bitit. Jäljelle jäävä arvo sisältää nyt vain halutut neljä bittiä, jotka ovat otsakkeiden pituuden sisältävä arvo. Tämän operaation jälkeen ohjelmisto tarkistaa, että verkkopaketin tyyppi on joko IPv4 tai IPv6, muussa tapauksessa ohjelmisto antaa virheen paketin yhteensopimattomuudesta ohjelmiston kanssa.

Seuraava tavu sisältää tietopalvelupistekoodin (DSCP) ja ruuhkautumisen ilmoituksen (ECN) datan, mutta ne ovat tässä sivuutettu, koska ne eivät datana ole ohjelman kannalta kovin olennaisia, joten ohjelma siirtyy suoraan eteenpäin. Seuraavat kaksi tavua sisältävät koko internet-paketin pituuden ilmaisevan datan ja ne luetaan ulos otsakkeista yhtenä isona arvona. Tätä seuraa iso hyppy otsakkeissa eteenpäin, sillä seuraavat neljä tavua sisältävät dataa, joka ei jälleen ohjelmalle ole olennaista. Ohjelma on nyt otsakkeiden tavuissa sijainnissa 8, joka sisältää paketin eliniän (TTL). Elinikä ilmaistaan sekunneissa ja sen ideana on estää verkkovirhettä, mikäli reitityssilmukka tapahtuu. Elinikä toimii periaatteella, jossa eliniän arvoa pienennetään yhdellä jokaisessa reitityshypyssä, eli elinikä samalla ilmaisee myös paketin maksimaalisen reitityshyppyjen määrän. Mikäli missään vaiheessa paketin reitittämistä elinikä saavuttaa nolla-arvon, paketin saava reititin sivuuttaa paketin.

Otsakkeista seuraavana on paketin protokollan ilmaisin. Internet-paketeilla on olemassa monta standardisoitua protokollaa, mutta niistä yleisimmät ovat TCP ja UDP. Tämän tietäminen on dataa lähettäessä ja vastaanottaessa olennaista, joten tämä tieto luetaan ylös. Otsakkeista seuraavana on verkkopaketin otsakkeiden checksum-hash, eli otsakkeista laskettu tiiviste. Saapuessa reitittimelle reititin laskee otsakkeista tiivisteen ja vertaa sitä mukana tulleeseen tiivisteeseen. Mikäli tulos on 0xffff, on data kunnossa ja reititin käsittelee paketin. Muussa tapauksessa reititin sivuuttaa paketin korruptoituneena. Tämä tieto ei ole ohjelmalle olennaista, koska reitittimet huolehtivat tiivisteen vertaamisesta ja laskemisesta, joten se on tässä sivuutettu.

Tämän jälkeen otsakkeiden datana ovat paketin lähde- ja kohdeosoitteet, kukin neljä tavua pitkiä. Myös jokainen näistä neljästä tavusta osoitetta kohden ovat erillisiä siten, että jokaisen tavun arvo ilmaisee eri osiota osoitteessa. Tästä datarakenteesta muodostuvat osoitteet ovat muodossa xxx.xxx.xxx.xxx, jossa jokainen pisteellä eroteltu tietue vastaa sitä vastaavan tavun arvoa. Datan purku osoitteille tapahtuu lukemalla koko osoitteen neljä tavua yhdeksi isoksi heksadesimaaliarvoksi ja antamalla se erilliselle apufunktiolle, joka jaottelee heksadesimaalin tavuiksi ja muuntamalla tavut numeraaliseen muotoon. Numeraaliset arvot varastoidaan väliaikaisesti listassa ja lopulta yhdistetään yhdeksi string-muuttujaksi, jossa arvot ovat eroteltu toisistaan pisteellä, muodostaen ihmiselle selkolukuisen IP-osoitteen. Tämä logiikka toteutetaan kummallekin osoitteelle ja molemmista rakennetut string-arvot varastoidaan omiin tietueisiinsa.

IPv4-pakettien otsakkeiden pituus ei ole vakio, vaan ne voivat myös sisältää lisätietoja otsakkeiden viimeisessä kentässä, options. Tämän kentän pituus ei myöskään ole vakio, vaan voi olla mikä vain tavumäärä nollan ja neljänkymmenen väliltä. Jotta otsakkeiden luku osataan lopettaa oikeassa kohdassa ja siirtyä paketin datan lukemiseen, tarvitaan avuksi aiemmin luettua IHL-arvoa, eli otsakkeiden pituuden ilmaisinta. Otsakkeiden pituuden arvo on välillä 5–15, jossa yksi arvo vastaa 4 tavua. Täten arvo 5 tarkoittaa 20 tavua, joka on otsakkeiden pituus ilman options-kenttää ja vastaavasti arvo 15 tarkoittaa otsakkeiden maksimaalista pituutta, 60 tavua. Tästä arvosta on täten helppo laskea mahdollisen options-otsakkeen pituus. Pituus saadaan kaavalla (IHL - 5) * 4. Mikäli arvo on suurempi kuin nolla, options on määritelty otsakkeissa ja se luetaan heksadesimaalina ulos.

Koska options on harvemmin käytettävä kenttä, mutta sitä ei voida kuitenkaan sivuuttaa, koska sen olemassaolo vaikuttaa datan sijaintiin paketissa, lukee ohjelma sen ulos, mutta varsinaista kentän sisältöä ohjelma ei pura, vaan se tarjotaan raakana heksadesimaalina. Lopulta ohjelma siirtyy itse dataan, jonka alkupisteen sijainti on nyt tiedossa. Datan loppupiste puolestaan saadaan koko datapaketin pituudesta. Näillä tiedoilla ohjelma lukee datan ulos raakana Buffer-tyypin datana.

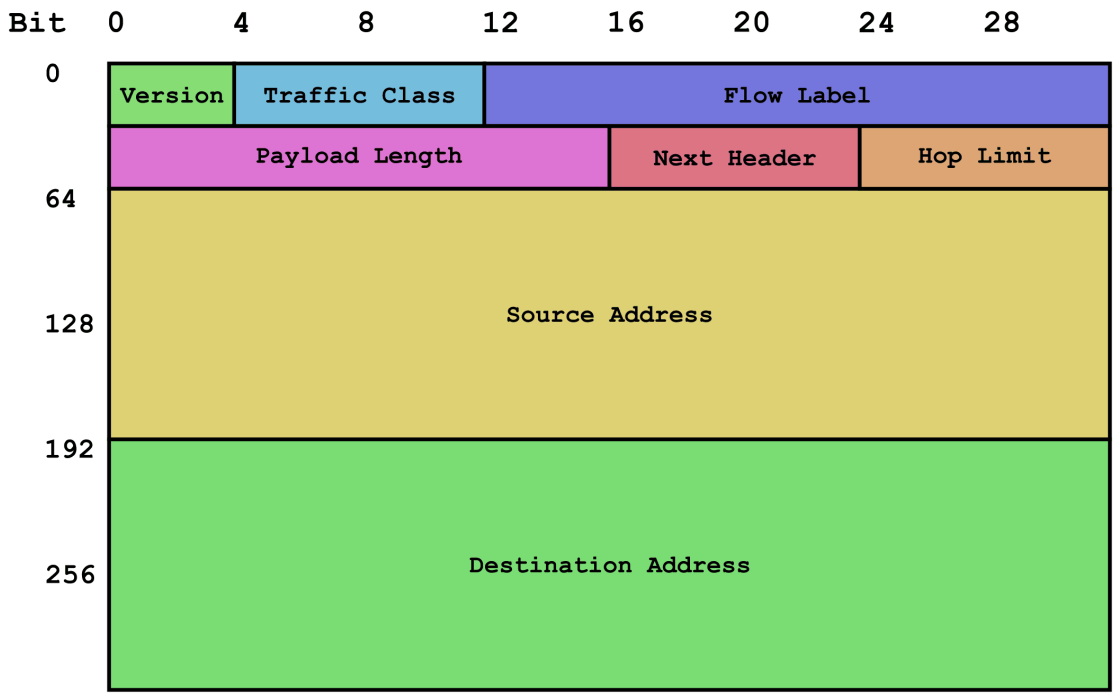
Otsakkeet ovat nyt purettu erillisiksi irrallisiksi tietueiksi ja kuten ethernet-kehyksissä, myös internet-paketilla on ohjelmassa olemassa oma datatyyppinsä, joten purettu data on aina samassa muodossa (kuvio 6). Ohjelma muodostaa tätä datatyyppiä noudattavan json-objektin ja asettaa siihen otsakkeista puretun datan, sekä payloadin ja palauttaa objektin return-funktion avulla takaisin kontekstiin, josta metodi kutsuttiin. Datatyyppi säilyy samana riippumatta internet-paketin versiosta.

```
interface InternetPackage {
  version: NetworkProtocol;
  ihl?: number;
  totalLength: number;
  protocol: IPProtocol;
  ttl: number;
  ipSource: string;
  ipDestination: string;
  options?: string;
  payload?: Buffer;
}
```

Kuvio 6: Internet-paketin puretun datan rakenne (Kuvankaappaus ohjelmiston lähdekoodista. 2024.)

IPv4-paketin purkajan toteuttamisessa aikaa vievin osuus oli ethernet-kehyksen tavoin datarakenteeseen perehtyminen. IP-osoitteiden purkaminen vaikutti alkuun haasteelliselta, mutta logiikan toteuttaminen apufunktioon selkeytti ajatusprosessia hyvin ja osoittautuikin lopulta helpommaksi, kuin alun perin vaikutti. Options-headerin olemassaolon ja sen mahdollisesti aiheuttaman payloadin sijainnin siirtymisen selvittäminen oli työläs, koska en heti sisäistänyt konseptia siitä, kuinka IHL:n arvo tulisi tulkita ja miten se vaikuttaisi tähän headeriin. Itse asiassa alkuun tämä headerin määrittäminen ei toiminutkaan odotetulla tavalla, mikä paljastuu myöhemmässä vaiheessa yksikkötestejä toteuttaessa. Kaiken kaikkiaan IPv4-pakettien purkamisen logiikan toteuttamiseen kului aikaa noin 24 tuntia. Tähän aikaan ei sisälly yksikkötesteihin mennyttä aikaa, eikä testeistä seurannutta buginkorjuuta, ne luetellaan myöhemmin.

IPv6-pakettien purkamisen logiikka on kirjoitettu samaan metodiin kuin IPv4-pakettien purku, mutta logiikat erotellaan toinen toisistaan paketin versio-otsakkeen datan avulla. Kuten tästä voidaan päätellä, samoin kuin IPv4-paketissa, IPv6-paketin purkaminen alkaa lukemalla versio-otsake. IHL-arvo asetetaan vakioarvoon 40, koska toisin kuin IPv4-paketeilla, IPv6-pakettien otsakkeet ovat aina samanpituiset (kuvio 7). Kuten IPv4-pakettien purkamisen osiossa selitettiin, tässä vaiheessa ohjelmisto tarkistaa verkkopaketin tyypin ja jatkaa suorittamista vain, jos paketin tyyppi on joko IPv4 tai IPv6.



Kuvio 7: IPv6-paketin rakenne bittitasolla (Networkel. 2024.)

IPv6-paketit sisältävät tämän jälkeen otsakkeet Traffic Class (TC) ja Flow Label (FL), mutta ne eivät ole ohjelmiston datan kannalta olennaisia, joten ne ovat sivuutettu. Koska yleisesti IPv6-pakettien otsakkeet ovat samantapaiset kuin IPv4-paketeilla, seuraavana vuorossa on paketin kokonaispituus. Toisin kuin IPv4:ssa, IPv6-paketissa luku ilmaisee vain itse payloadin pituuden, ei koko paketin pituutta. Otsakkeen tavupituus on kuitenkin sama, kaksi tavua, joten samaa logiikkaa IPv4-pakettien kanssa voidaan käyttää uudelleen, mutta siihen täytyy vain jälkikäteen lisätä otsakkeiden vakiopituus 40 tavua.

IPv4-paketeista poiketen seuraavana otsakkeena on Next Header, joka ilmaisee paketin käyttämän protokollan samalla periaatteella kuin IPv4-paketeissa. Tämäkin on pääasiassa joko TCP tai UDP. Kuten IPv4-paketeissa, otsakkeen data puretaan talteen. Käänteisesti IPv4-paketteihin nähden seuraavana otsakkeena on IPv6:n versio TTL:stä (kuvio 5). TTL:n toimintaperiaate selitettiin IPv4-pakettien purkamisessa, kuinka käytännössä sillä ilmaistaan reitityshyppyjen maksimaalinen määrä. IPv6-paketeissa tämä otsake onkin nimetty nimellä Hop Limit (suomeksi hyppyraja), joka nimensä mukaisesti ilmaisee paketin maksimihyppymäärää (kuvio 6). Samoin kuin IPv4:ssä, hyppymäärää vähennetään jokaisessa reitityshypyssä yhdellä ja mikäli paketti ei ole kohdeosoitteessa ja hyppymäärä saavuttaa arvon nolla, paketin vastaanottava reititin sivuuttaa paketin virheellisenä välttääkseen reititysvirheen. Otsakkeen pituus on myös sama, yksi tavu, ja se luetaan talteen.

Otsakkeita on enää tässä vaiheessa jäljellä kaksi kappaletta, mutta molemmat niistä ovat isoja tavumäärältään. Kyseessä ovat paketin lähde- ja kohdeosoitteet. Toisin kuin IPv4-paketeissa, jossa osoitteen pituus on 32 bittiä, eli neljä tavua, IPv6-osoitteen pituus on 128 bittiä, eli 16 tavua. Myös osoitteen rakenne on erilainen: IPv4-osoitteet koostuvat neljästä tavun kokoisesta segmentistä, mutta IPv6-osoitteet puolestaan koostuvat kahdeksasta kahden tavun kokoisesta segmentistä. Tämän vuoksi osoitteiden purkamiseen toteutettiin erillinen apufunktio. Kuten IPv4-osoitteilla, osoite luetaan ensin kokonaisena raakana heksadesimaalina ulos ja annetaan sen jälkeen apufunktiolle parsittavaksi. Apufunktio pilkkoo heksadesimaalin kahden tavun paloihin, mutta toisin kuin IPv4-osoitteiden purkaja, tämä ei muunna heksadesimaalia numeraaliseen muotoon, koska IPv6-osoitteet esitetään heksadesimaalimuodossa. Pilkkominen toteutetaan funktion paikalliseen väliaikaiseen listaan ja lopulta listan sisältämät heksadesimaalimerkkijonot yhdistetään toisiinsa käyttäen puolipistettä (:) erottimena, noudattaen yleistä IPv6-osoitteiden esitysmallia.

Osoitteet voisi myös lyhentää, mikäli ne sisältäisivät yhtenäisen jonon nollia ja tämä ilmaistaisiin kahdella peräkkäisellä kaksoispisteellä (::), esimerkiksi fe80::c555:f56:6f96:31d1. Kuitenkin osoite voi sisältää vain yhden tällaisen lyhenteen. Tämä lyhenne on kuitenkin täysin valinnainen, jonka takia tällaisten lyhenteiden toteuttamiseen tarvittavaa logiikkaa ei ohjelmistoon ole lisätty. Kaikki tämäntapainen lisäprosessointi hidastaisi paketin datan purkamista ja se ei kuitenkaan tarjoaisi varsinaista hyötyä ohjelmiston toiminnan kannalta. Kun osoite on paloiteltu ja rakennettu yleiseen esitysmuotoon, palautetaan muodostettu osoite takaisin funktion kutsujalle, missä osoite otetaan talteen omaan muuttujaansa. Tämä sama prosessi toteutetaan sekä lähde- että kohdeosoitteelle.

IPv6-paketit eivät sisällä valinnaisia kenttiä, kuten options-kenttä IPv4-paketissa, vaan otsakkeet loppuvat aina kohdeosoitteeseen, tavuun numero 40. Tämän jälkeen loppuosa paketista on vain itse paketin payloadia, joka on helppo ottaa erikseen leikkaamalla paketin bittijonon sisältö tavusta 40 eteenpäin aina jonon loppuun saakka. Tämä erikseen leikattu bittijono sisältää nyt pelkän IP-paketin payloadin ja se otetaan talteen.

Kuten IPv4-paketeilla, puretusta datasta muodostetaan tyyppivarma json-objekti (kuvio 6), joka sisältää paketin version, otsakkeiden pituuden, paketin kokonaispituuden, paketin datan protokollatyypin, paketin eliniän, eli reitityshyppyjen maksimimäärän, kohde ja lähde IP-osoitteet, mahdollisen options-otsakkeen, sekä lopulta itse paketin payloadin. Objekti muodostetaan käyttäen samaa koodia kuin IPv4-paketeilla ja palautetaan samalla tavalla return-funktiolla takaisin metodikutsun tehneeseen kontekstiin. Kontekstissa tästä puretusta datasta voitaisiin edelleen eritellä payload, joka annettaisiin korkeamman tason purkajalle käsiteltäväksi, mikä mahdollistaa helpon jatkokehittämisen. Tässä työssä sitä ei kuitenkaan tehdä.

Toisin kuin IPv4-paketinpurkamisessa, IPv6 osoittautui huomattavasti yksinkertaisemmaksi ja nopeammaksi tehdä, koska toiminnallisuus on rakennettu samaan koodiin kuin IPv4, mutta siirtää vain paketinpurkajan ”offset”-muuttujaa eri tahtiin kuin IPv4:ssä paketin rakenteellisen eron takia. Lisäksi näissä paketeissa ei tarvitse huomioida valinnaisia otsakkeita ja niistä aiheutuvaa vaihtelua otsakkeiden kokonaispituudesta. Merkittävin huomioitava asia olikin TTL:n ja protokollan käänteinen järjestys IPv4:ään nähden. Vastaavasti myös osoitteiden erilaisen rakenteen ja pituuden vuoksi osoitteet prosessoivasta apufunktiosta piti toteuttaa oma versionsa IPv6-osoitteille. Tämän apufunktion logiikka on kuitenkin yksinkertaisempi osoitteiden heksadesimaalisen esitystavan takia, joten se oli yksinkertainen toteuttaa. Tässä vaiheessa IPv6-pakettien purkajan toteuttamiseen oli kulunut aikaa yhteensä noin 10 tuntia, mikä on merkittävästi vähemmän kuin IPv4-osoitteilla. Merkittävä osa tästä ajasta kuluikin IPv6-osoitteen rakenteen tulkintaan ja sen vaativien muutostöiden suunnitteluun. Varsinainen toteutusvaihe oli huomattavasti nopeampi kuin IPv4:ssä.

6.4 Yksikkötestit

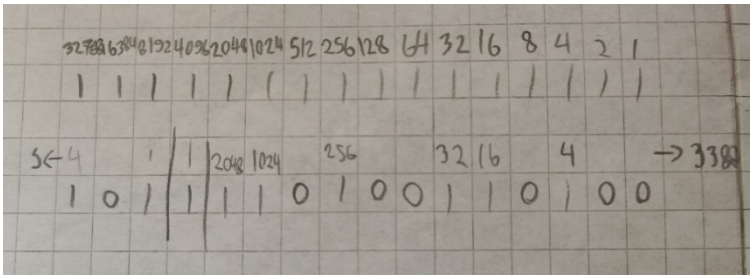
Ohjelmisto kykeni tässä vaiheessa purkamaan sekä ethernet-kehyksiä, että IP-paketteja. Verrattuna datarakenteeseen mitä referenssimateriaaleista löytyi, data vaikutti olevan kunnossa, samoin esimerkiksi ping-työkalulla luodun verkkoliikenteen data vaikutti tulevan oikeanlaisena ulos. Kuitenkin on tärkeää, että ohjelmalle on toteutettu kattavat yksikkötestit, jotta ohjelman toiminta voidaan varmistaa, etenkin jos koodia muokataan myöhemmin. Täten jos jokin toiminnallisuus rikkoontuu tai datan purku korruptoituu, virheen aiheuttama regressio ilmenee yksikkötesteissä ennen kuin tämä virheellinen toiminnallisuus päätyy tuotantoympäristöihin asti.

Kuten ohjelmiston kirjoittamisessa, aloitin testien kirjoittamisen ethernet-kehyksille. Testaamisessa päätavoitteena on tarkistaa, että yksittäiset otsakkeet puretaan paketista ulos oikein. Vastaavasti mikäli paketti on liian lyhyt, paketti tulee sivuuttaa virheellisenä mahdollisimman aikaisessa vaiheessa, jotta vältetään turha prosessointi.

Jotta paketinpurkajaa voidaan testata, täytyy sille saada syötettyä tunnettua dataa sisältävä raaka bittijono, joka on Typescriptissä tietotyypiltään Buffer, koska paketinpurkaja odottaa datan olevan siinä muodossa. Tätä varten täytyi toteuttaa erillinen datageneraattori. Generaattorin toiminnallisuutta toteuttaessa täytyi olla tarkkana, että sen toimii täsmälleen odotetulla tavalla. Tähän käytin apuna oikeaa ethernet-kehystä ja sen dataa. Tämä ethernet-kehys oli luotu käyttäen linuxin ping-komentoa. Tämän avulla datasta tiedettiin sen lähde ja kohde MAC-osoitteet, kuten myös datan EtherType, joka on IPv4. Datapaketti ei myöskään sisältänyt VLAN-tunnusta. Tavoitteena oli tehdä datageneraattori ja toteuttaa se siten, että sen generoiman paketin sisältämien otsakkeiden osuus bittijonossa vastaa oikean datapaketin otsakkeiden bittijonoa, eli generaattori matkii oikean datapaketin rakennetta. Tämän pohjalta generaattorin toteuttaman datan pitäisi olla yhdenmukaista oikean datan kanssa. Generaattori toimii periaatteella, että mikäli syöteparametreja ei anneta, generaattori luo datan satunnaisesti, mutta kuitenkin standardin mukaisesti. Vastaavasti generaattori ottaa vastaan minkä vaan paketin otsakkeen syöteparametrina, mikä mahdollistaa helposti halutun datan sisältävän datapaketin generoimisen.

Kun generaattori oli valmis, alkoi itse varsinainen testien toteuttaminen. Ensimmäisenä toteutin testauksen MAC-kohdeosoitteelle. Annoin generaattorille halutun MAC-osoitteen parametrina ja sen jälkeen annoin generoidun paketin paketinpurkajalle. Paketinpurkajalta takaisin tulevasta puretusta dataobjektista luetaan MAC-kohdeosoite ja osoitetta verrataan aiemmin määritettyyn osoitteeseen. Mikäli osoitteet ovat samat, on data purettu ulos oikein ja testi on läpäisty hyväksytysti. Sama toteutetaan MAC-lähdeosoitteelle.

Seuraavaksi testit toteutettiin VLAN-tunnuksen purkamiselle. Aivan alkuun täytyi luoda bittitasolla oikeanmuotoinen VLAN-tunnus 802.1Q-standardia (IEEE 2003.) noudattaen (kuvio 3). Bittijono oli helpoin hahmotella ensin paperille (kuva 1), jonka jälkeen muunsin jonon heksadesimaalimuotoon. Tämä heksadesimaali puolestaan asetettiin testiin syöteparametriksi VLAN-tunnukselle ja ensin testataan, että paketinpurkaja havaitsee tunnuksen olemassaolon. Mikäli kyllä, testi on läpäisty. Seuraavassa testissä testattiin tunnuksen datan purkamista. Testissä käytettiin samaa heksadesimaalia kuin aiemmassa testissä, mutta tässä sen bittijonosta käsin laskettua dataa verrattiin paketinpurkajan tulokseen. Mikäli tulos vastaa odotettua dataa, on testi läpäisty onnistuneesti.



Kuva 1: Hahmotelmaa VLAN-tunnuksen bittijonosta (prioriteetti 5, CFI 1 ja VLAN-ID 3380) (Kuva: Niko Huuskonen. 2024.)

Seuraavaksi testattavana oli paketin payloadin, eli kapseloidun datan purkaminen. Tätä varten luotiin paketti, jolla on tunnettu datan sisältö. Tämä data ei vastannut ylemmän kerroksen formaatin mukaista dataa, mutta se ei vaikuta ohjelmiston toimintaan. Testi on hyväksytty, mikäli paketinpurkajan tuloksen payload vastaa ennalta määritettyä arvoa, kun se muutetaan takaisin string-muotoon.

Viimeisenä testattavana asiana oli virheellisen ethernet-kehyksen tunnistaminen. Paketinpurkaja laskee läpi epäkelvolliset arvot eri otsakkeissa, mutta mikäli kehyksen pituus on liian lyhyt, purkaja heittää virheen metodin kutsujalle. Testissä purkajalle annetaan tarkoituksellisesti liian lyhyt ethernet-kehys ja mikäli virhe heitetään, on testi hyväksytty.

Ethernet-kehysten testien toteuttamisen jälkeen siirryin heti toteuttamaan samantapaisia yksikkötestejä internet-pakettien purkajalle. Sekä IPv4-, että IPv6-paketit testataan erikseen. Kuten ethernet-kehyksien kanssa, IP-paketeille luotiin oma generaattorinsa. Ainoa pakollinen ennalta määriteltävä kenttä generaattorille on paketin versio, joko 4 tai 6. Tämän tiedon pohjalta funktio luo annettua versiota vastaavan satunnaista dataa sisältävän verkkopaketin. Tämäkin generaattori ottaa vastaan myös määrityksiä eri paketin kentille, mikä mahdollistaa tiettyjen paketin otsakkeiden tai paketin datan sisällön määrittämistä halutuun arvoon. Kun generaattori oli valmis, aloitin testien tekemisen IPv4-formaatille.

Aloitin testien tekemisen otsakkeiden mukaisessa järjestyksessä, sivuuttaen kuitenkin paketin version otsakkeen, koska testeissä paketin versio on ennalta määritetty ja mikäli paketinpurkaja lukisi version väärin, tämä johtaisi testin epäonnistumiseen koska paketin dataa yritettäisiin lukea väärästä kohdasta ja paketinpurkaja hyväksyy vain versiotunnukset 4 ja 6. Täten ensimmäisenä testattavana asiana olikin paketin TTL, eli elinikä. Tätä varten testi luo IPv4-datapaketin, jolla on ennalta määritetty tunnettu elinikä. Tämän jälkeen paketti annetaan paketinpurkajalle ja purkajalta tulleesta datasta saatua elinikää verrataan testin alussa määriteltyyn elinikään. Mikäli arvot täsmäävät, on testi hyväksytty.

Seuraavana otsakkeena vuorossa oli paketin protokolla ja testauksessa noudatettiin samaa menetelmää kuin eliniän kanssa. Jos ennalta määritetty arvo vastaa paketinpurkajan tuloksissa olevaa arvoa, on testi hyväksytty. Tämän jälkeen testattavana olivat lähde- ja kohde-IP-osoitteet. IP-osoitteet annetaan generaattorille muodossa ”xxx.xxx.xxx.xxx”, voidaan antaa myös ilman numeron alussa olevia mahdollisia nollia, generaattori osaa huomioida ne pois, kun osoitteen lohkot muutetaan generaattorin sisällä numeraaliseen muotoon ennen kuin niistä muodostetaan varsinainen heksadesimaaliarvo, joka asetetaan pakettiin. Kuitenkin itse vertausoperaatiossa näitä ylimääräisiä numeron alussa olevia nollia ei saa olla, koska paketinpurkajan tuloksessa niitä ei ole, mikä johtaisi testin epäonnistumiseen. Options-otsakkeen purkamista ei tässä testata, koska ot-

sakkeen sisältöä paketinpurkaja ei muutenkaan käsittele, mutta jatkokehitysmielessä tähänkin olisi hyvä toteuttaa jonkinlaisia yksikkötestejä, viimeistään silloin, kun paketinpurkajaan lisättäisiin options-otsakkeen datan prosessointi. Viimeinen testi IPv4-paketille oli paketin dataosuuden purkaminen, eli payloadin purku raa'aksi bittijonoksi. Testattava data voi olla mitä vain, koska paketinpurkaja ei käsittele dataosuuden sisältöä, vaan antaisi sen seuraavan kerroksen purkajalle. Täten testissä data onkin vain yksinkertainen UTF-8 koodauksella oleva ”Hello World!” -teksti, joka on muutettu generaattorissa Buffer-tyyppimuotoon, eli bittijonoksi. Kun paketinpurkaja on purkanut datan, muutetaan se takaisin tekstimuotoon käyttäen aiemmin mainittua UTF-8 koodausta. Mikäli tekstit ovat samat, on testi hyväksytty. Tässäkin olisi hyvä olla rajatapaustesti myös mahdollisen options-otsakkeen kanssa siltä varalta, jos otsakkeen olemassaolo vaikuttaa dataosuuden lukemiseen.

IPv4-testien jälkeen olikin IPv6-pakettien purkamisen testaaminen. Testit noudattavat pääasiassa samaa toimintaperiaatetta kuin IPv4-pakettien purkamisessa, vaikka pakettirakenne onkin bittitasolla erilainen. Testit paketin protokollalle ja eliniälle ovat käytännössä identtiset. IP-osoitteiden testaaminenkin on pitkälti sama, mutta on tärkeä tietää, että osoite pitää antaa generaattorille kokonaisena ja sen pitää sisältää kaikki osoitelohkot, vaikka ne olisivatkin täynnä nollia. Syynä tähän on se, että generaattorissa ei ole logiikkaa lyhennettyjen osoitteiden käsittelyyn. Lisäksi tämä myös helpottaa annetun osoitteen vertaamista tulokseen, koska myös paketinpurkaja antaa osoitteen täydessä pituudessa sisältäen myös kaikki nollat.

Kun pakettien versiokohtaiset testit olivat tehty, testasin vielä viimeisenä, että paketinpurkaja antaa virheilmoituksen, mikäli paketin versio on jokin muu kuin 4 tai 6, jotka ovat ainoat paketinpurkajan tukemat versiot. Mikäli virhe annetaan ja funktion suoritus keskeytetään, on testi hyväksytty. Tässä vaiheessa koin IP-pakettien purkamisen testaamisen välttävän kattavaksi. Jatkokehitysmielessä yksi ensimmäisistä asioista olisi kuitenkin testien laajentaminen, jotta mahdolliset rajatapaukset katettaisiin mahdollisimman hyvin. Kuitenkaan mitään kriittistä, ohjelmiston hallitsemattomaan kaatumiseen johtavaa tilannetta ei pitäisi päästä

muodostumaan virheellisen datan kautta. Kaikkiaan niin ethernet-kehyksien, kuin IP-pakettien purkajien testien toteuttamiseen kului aikaa noin 12 tuntia, johon sisältyy niin testien, kuin myös pakettigeneraattoreiden toteuttaminen.

6.5 OSI-mallin korkeampien kerroksien purkaminen

Kuten aiemmin eritelty, OSI-mallissa on useita kerroksia verkkokerroksen päällä (kuvio 1). Tällä hetkellä paketinpurkaja käsittelee verkkopinon ns. mediakerrokset 2 ja 3, jotka ovat alimmat kerrokset, mihin PCAP pääsee käsiksi. Pinon seuraava kerros, kuljetuskerros (englanniksi ”transport layer”) määrittelee tasolla 3 määritellyn protokollan päällä kulkevan datan protokollasta. Kuitenkin nykyaikaan suurin osa tästä datasta on jonkinlaisen salauksen, useimmiten TLS-salaus takana, jotta yhteys on päästä-päähän salattu. Jotta ylempien kerroksien datanpurku olisi mahdollisimman hyödyllistä, täytyisi ohjelmistoon toteuttaa tällä kerroksella mahdollisen salauksen purkaminen, mikä on haasteellista toteuttaa etenkin, jos salauksessa käytettyjä avaimia ei tunneta. Tästä syystä opinnäytetyön kontekstissa ei mennä verkkodatan purkamisessa tätä syvemmälle, vaan ohjelmisto kattaa vain mediakerroksien datan purkamisen. Ohjelmisto on kuitenkin toteutettu siten, että se mahdollistaa jatkokehittämisen myös näille ylemmille kerroksille, vaikka niihin ei tässä työssä syvennytä. On totta, että esimerkiksi HTTP- ja FTP-protokollat ovat salaamattomia (Fielding & Nottingham & Reschke 2022) (Postel & Reynolds 1985), mutta näistäkin on olemassa salatut versiot ja suurin osa nykyliikenteestä tapahtuukin näillä salatuilla versioilla HTTPS ja FTPS (käyttävät TLS-salausta) tai SFTP, joka käyttää SSH-protokollaa datan tunnelointiin.

6.6 Ohjelmiston API:n rakentaminen

Tässä vaiheessa ohjelmiston rakenne oli aika sekava. Koko ohjelmistoa piti kasassa sen index.ts-tiedosto, johon oli vain luotu funktiot PCAP-yhteyden avaamiselle ja sen tarjoaman ”raw”-tapahtumapyynnön käsittelylle. Tämä funktio

puolestaan ajoi eri kerroksien paketinpurkajat ja lopulta vain tulosti kerroksista puretut tiedot konsoliin. Varsinaista selkeää ja helppokäyttöistä rakennetta ohjelmiston käyttämiseen ja jatkokehittämiseen ei siis ollut, vaikkakin se onkin ohjelmiston käyttötarkoituksen kannalta olennainen.

Mietittyäni eri toteutusmalleja päädyin lopulta luokkapohjaiseen ratkaisuun, koska se vaikutti olevan helpoin ja yksinkertaisin toteuttaa. Tässä ratkaisussa nykyinen index.ts-tiedoston sisältö siistittäisiin rakenteellisesti modulaarisemmaksi ja helpompilukuiseksi, sekä siitä tehtäisiin helposti myöhemmin laajennettava. Tämä uusi koodi puolestaan sijoitettaisiin kokonaisuudessaan uuden Typescript-luokan sisälle olio-ohjelmoinnin periaatteiden mukaisesti ja luokkaan toteutettaisiin metodeja, joiden kautta luokan toimintaa hallitaan. Luokka tarjoaa myös tapahtumapohjaisen metodin ”on”, jonka ensimmäinen parametri määrittelee odotettavan tapahtuman nimen. Nimen voi kirjoittaa joko tekstinä, tai sen voi myös antaa ohjelmistoon tehdyn enum-tyyppisen muuttujan kautta. Seuraava parametri metodissa on metodin takaisinkutsufunktio. Tämä voi olla viite muualla sijaitsevaan funktioon, joka vastaa syöteparametreiltaan takaisinkutsufunktion rakennetta tai funktio voi olla myös suoraan toteutettuna toisen parametrin sisälle. Luokka antaa takaisinkutsufunktioon tapahtumatyypin mukaan dataa syöteparametrina. Luokan toiminta aloitetaan olio-ohjelmoinnin periaatteiden mukaisesti luomalla siitä uuden olion ja sen jälkeen ajamalla luokan tarjoamia metodeja oliossa. Tähän kuitenkin syvennytään yksityiskohtaisemmin tuloksissa ja demonstraatiossa. Aikaa API:n toteuttamiseen suunnittelu mukaan lukien kului noin vajaa 10 tuntia, mistä suurin osa ajasta meni tyyppivarman tapahtumapohjaisen toiminnallisuuden toteuttamiseen.

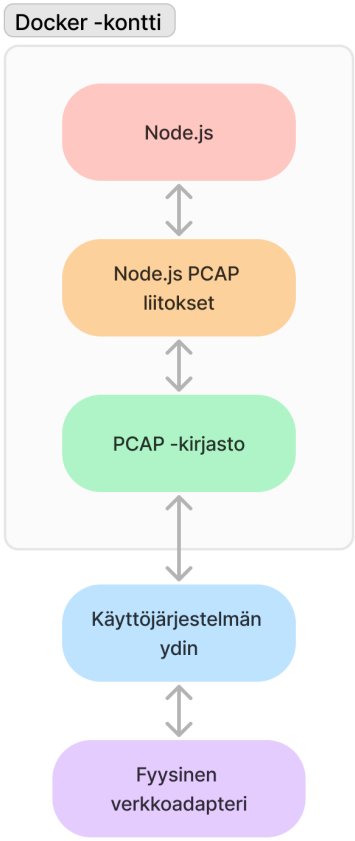
7 Tulokset ja ohjelmiston demonstrointi

Ohjelmiston varsinainen kehitystyö tälle opinnäytetyölle on täten saatu päätökseen. Kehitystyö tapahtui melko laajalla aikaikkunalla ja usein siten, että kun ke-

hitystä tapahtui, niin sitä tapahtui kerralla paljon. Kehitysajankohdat määräytyivät pitkälti aikataulujen ja oman jaksamisen mukaan, mutta lopulta ohjelmisto on saatu pisteeseen, jossa se voidaan määritellä tämän opinnäytetyön kannalta valmiiksi. Tässä osiossa käydään läpi, millainen lopputulos kehitysprosessista valmistui, miten ohjelmistoa voidaan jatkokehittää ja miten ohjelmistoa käytetään. Ohjelmiston käytöstä esitellään myös hyvin yksinkertainen demonstraatio, jossa samalla havainnollistetaan ohjelmiston toimivuus.

Kuten aiemmin eriteltiin, ohjelmiston tavoitteena on lukea reaaliaikaista verkkodataa sille annetusta verkkoportista PCAP-API:a hyödyntäen ja purkaa sieltä irti tietoa. Tämä tieto tarjotaan ohjelmistoa käyttävälle kehittäjälle, jotta kehittäjä voi toteuttaa omaa toiminnallisuuttaan ohjelmiston päälle. Ohjelmiston tulee myös toimia Docker-kontitusohjelmiston sisällä. Ideana tällä on mahdollistaa mahdollisimman helppo ja vaivaton käyttöönotto erilaisissa suoritusympäristöissä, kuten palvelimella ajettavassa ohjelmistopinossa.

Näistä tavoitteista valmis ohjelmisto suoriutuu riittävästi. Ohjelmisto voisi purkaa enemmänkin dataa, mutta kuten aiemmin selitettiin, tämä vaatisi suurempaa logiikkaa mahdollisen kryptauksen purkamiselle, mikä monimutkaistaisi ohjelmistoa suuresti. Tästä syystä toteutuksessa tyydyttiin OSI-mallin kerroksiin 2 ja 3, eli mediakerroksiin. Tasoon 1 myöskään ei valitettavasti päästy, koska PCAP:lla ei ole pääsyä niin matalan tason dataan. Tasosta 2 ja 3 ohjelmisto purkaa kaikki olennaisimmat tiedot ylös ja tarjoaa niille myös tyypitykset.

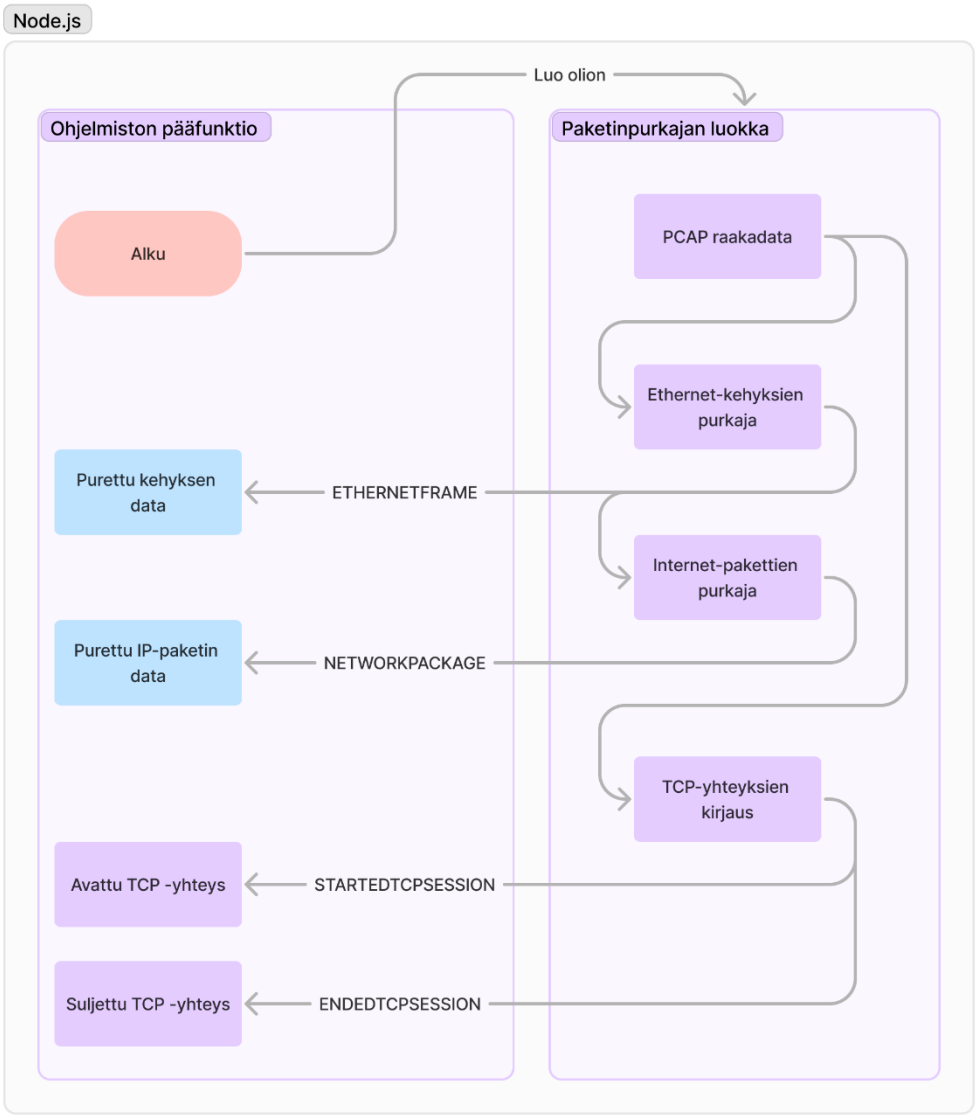


Kuvio 8: Ohjelmiston rakenne ja asema tietokonejärjestelmässä (Niko Huuskonen. 2024.)

Suorituskyvyn kannalta ohjelmisto on hyvä, etenkin kun otetaan huomioon, että toteutus on tehty Typescriptillä. Merkittävät tekijät tässä ovat tapahtumapohjainen toimintamalli ja se, että itse varsinainen PCAP-rajapinta on toteutettu C-ohjelmointikielellä ja rajapintaa vain hallinnoidaan Typescriptin päästä. Tapahtumapohjainen toimintamalli varmistaa, että datan prosessointi alkaa välittömästi datapaketin saapuessa ja vastaavasti kehittäjä saa puretun datan välittömästi käsiteltäväkseen, kun purkaminen on valmistunut. Ohjelmiston rakenteen pitäminen yksinkertaisena ja modulaarisena edesauttaa suorituskyvyn pitämisenä ideaalisena ja myös samalla helpottaa mahdollisten virhetilanteiden diagnosointia, koska virhe on helppo rajata johonkin tiettyyn moduuliin. Yllä oleva kuvio (kuvio 8) esittää ohjelmiston rakenteen ja aseman järjestelmässä, jossa ohjelmistokonttia suoritetaan.

Ohjelmistolle merkittävää on se, että se on myös helposti jatkokehitettävissä. Tätä varten ohjelmisto on rakennettu mahdollisimman laajennettavaksi ja kaikki olemassa oleva toiminnallisuus on koostettu yhdeksi isoksi ohjelmointirajapinnaksi Typescriptin luokkarakennetta hyväksikäyttäen. Typescriptin tyyppivarmuutta hyväksikäyttäen kyseinen luokka, kuten myös ohjelmisto yleisesti on tehty tyyppivarmaksi. Tästä on kaksi merkittävää hyötyä: Kehittäjä tuntee funktioiden syöteparametrien vaatimat datatyypit ja tuntee myös funktiosta tulevan datan tyypityksen sekä kehittäjän ohjelmistonkehitysympäristö osaa automaattisesti ennakoida paremmin, mitä kehittäjä voi mahdollisesti olla aikeissa tehdä ja tarjota vain kelvollisia vaihtoehtoja tyypitettyihin kenttiin. Tyyppivarmuus yleisesti tekee ohjelmistonkehittämisestä varmempaa ja helpompaa, sekä minimoi mahdollisten ajonaikaisten tyyppivirheiden tapahtumisen.

Yksittäisten moduulien toiminta ja rakenne on jo selitetty selkeämmin aiemmissa osissa, joten tässä keskitytään enemmän lopputulokseen ja sen käyttöön, mukaan lukien mahdollisen jatkokehittämisen. Paketinpurkajan rakenteesta löytyy myös kuvio (kuvio 9), joka voi selkeyttää rakenteen ymmärtämistä. Hyvä aloituskohta on itse varsinainen datanpurkajan pääluokka, sen rakenne, tyypitys ja mahdollinen laajentaminen. Paketinpurkajan pääluokka voidaan jaotella karkeasti kolmeen osaan: Luokan luontirakenteeseen ja käynnistysmetodiin, pakettien purkajiin sekä luokan tiedonvälittämiseen tapahtumapohjaisilla takaisinkutsufunktioilla. Perehdytään näihin kehityskokemuksen näkökannasta.



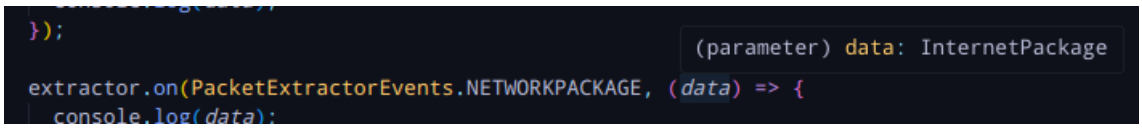
Kuvio 9: Ohjelmistopohjan rakenne yksinkertaistettuna (Niko Huuskonen. 2024.)

Kehittäjän kannalta olennaisinta luokkarakenteessa on ymmärtää luokan käyttötapa ja selkeä koodin järjestely: mikä tekee, mitä tekee, miksi tekee ja milloin tekee. Koska kyseessä on olioluokka, aloituskohtana toimii uuden olion luonti luokasta. Olio vaatii syöteparametrina ainakin verkkoportin nimen, mutta sille voidaan antaa myös lisäparametreja. Kun uusi olio on luotu, kannattaa tässä vaiheessa määritellä olion tapahtumien kuuntelufunktiot ja niiden sisältämät takaisinkutsufunktiot. Kuvio 9:stä käy ilmi toiminnan kannalta olennaisimmat tapahtumakutsut, mutta luokka tarjoaa myös kutsut PCAP-yhteyden avaamiselle ja sulkeutumiselle sekä kaikille suorituksen aikana mahdollisesti tuleville virheille, jotka luokkarakenne nappaa paketin purkuvaiheessa. Viimeisenä suoritetaan

luokan toiminnan käynnistävä funktio ’monitor’. Alla oleva kuvio (kuvio 10) esittelee yksinkertaisimman version ohjelmiston luokan käyttämisestä siten, että kaikki luokan tapahtumakutsut ovat käytössä.

```
const networkInterface = process.env.INTERFACE;

if (!networkInterface)
  throw new Error('Network interface is not defined, check your .env file!');

const extractor = new PacketExtractor(networkInterface, {
  isTcpTrackerEnabled: true,
});

extractor.on(PacketExtractorEvents.OPEN, () => {
  console.log("Opened PCAP session");
});

extractor.on(PacketExtractorEvents.CLOSE, () => {
  console.log("Closed PCAP session");
});

extractor.on(PacketExtractorEvents.ETHERNETFRAME, (data) => {
  console.log(data);
});

extractor.on(PacketExtractorEvents.NETWORKPACKAGE, (data) => {
  console.log(data);
});

extractor.on(PacketExtractorEvents.STARTEDTCPSESSION, (data) => {
  console.log(data);
});

extractor.on(PacketExtractorEvents.ENDEDTCPSESSION, (data) => {
  console.log(data);
});

extractor.on(PacketExtractorEvents.ERROR, (data) => {
  console.error(data);
});

extractor.monitor();
```

Kuvio 10: Esimerkki, jossa olio luodaan ja kaikkia olion tarjoamia tapahtumakutsuja käytetään (Niko Huuskonen. 2024.)



```
});

(parameter) data: InternetPackage

extractor.on(PacketExtractorEvents.NETWORKPACKAGE, (data) => {
  console.log(data);
```

Kuvio 11: Datan tyypin etukäteen tietäminen kursorilla osoittaessa hyödyntäen kehitysympäristön kielipalvelinta (Niko Huuskonen. 2024.)

Kuviosta 10 käy hyvin ilmi, että osa tapahtumakutsuista tarjoaa myös dataa takaisinkutsufunktioon. Tämä data on tyyppivarmaa ja sen käyttö on valinnaista, luokka on rakennettu siten, että se kehitysympäristöjen Typescript-kielipalvelin osaa automaattisesti tapahtumatyypin perusteella määrittää, onko takaisinkutsufunktiossa dataa syöteparametreina vai ei. Tämä tulee hyvin ilmi kuviossa 11, jossa tietokoneen hiirikursori osoittaa dataan, mikä luo kuviossa näkyvän konteksti-ikkunan. Periaatteessa ohjelmisto voidaan rakentaa mieleisekseen ja käyttötarkoituksen mukaiseksi näitä jo olemassa olevia tapahtumia ja niiden tarjoamaa dataa käyttäen. Demopohja tarjoaa myös hyvän lähtökohdan, koska periaatteessa kehittäjän tarvitsee vain toteuttaa haluamansa takaisinkutsufunktiot nykyisten lokimerkintöjen paikalle.

Mikäli kuitenkin ohjelmiston ydintoiminnallisuutta halutaan laajentaa, on tämä täysin mahdollista ja se on otettu huomioon ohjelmistopohjaa toteuttaessa. Korkeampien verkkokerrosten purkamisen toteutus on helppoa toteuttaa olemassa olevien kerrosten purkajien päälle. Tämä edellyttää kuitenkin myös aiemmin mainitun kryptauksen huomioinnin ja käsittelyn. Seuraavaksi esittelen luokkarakenteen olennaisimmat elementit ja tyypitykset, jotta luokan jatkokehittäminen ja verkkokerrosten lisäys ja tyypittäminen olisi mahdollisimman yksinkertaista.

Luokan tapahtumat ovat tyypitetty luokkarakenteen ohessa olevan rajapinnan avulla käyttäen avain-arvo-pareja. Mikäli luokkaan lisätään uusia tasoja tai muuta toiminnallisuutta ja niille halutaan lisätä omat tapahtumat, täytyy tapahtumatyypit ja niiden mahdolliset takaisinkutsufunktion sisältämät datatyypit määritellä tähän rajapintaan. Tämän pohjalta ohjelmisto osaa itse automaattisesti rakentaa tyyppivarmuuden ja automaattisen täytön tuen uudelle tapahtumalle, joka helpottaa kehitysprosessin sujuvuutta ja minimoi virhealttiutta. Luokan tapahtumakutsujen käsittelyssä ja toteutuksessa hyväksikäytetään paljon Typescriptin tarjoamaa geneeristä muuttujatyyppiä, joka saa tyyppiarvon sen mukaan, minkä syöteparametrin geneerinen muuttuja saa.

```
public on<K extends Event>(
  event: K,
  listener: PacketExtractorCallback<EventMap[K]>
) {
  if (!this.eventListeners[event]) {
    this.eventListeners[event] = [];
  }
  this.eventListeners[event]?.push(listener);
}

private emit<K extends Event>(event: K, data?: EventMap[K]) {
  if (this.eventListeners[event]) {
    for (const listener of this.eventListeners[event]!) {
      listener(data as EventMap[K]);
    }
  }
}
```

Kuvio 12: Luokan sisäisen tapahtumakutsun luova emit-funktio ja tapahtuman oliosta ulos tuleva on-funktio

Kuten yllä olevasta kuvaajasta 12 voimme nähdä, molemmat metodit käyttävät geneeristä muuttujaa (K), jonka tunnetaan pohjautuvan tyyppiin Event. Event puolestaan on osa rajapintaa EventMap. Tämä mahdollistaa sen, että jos EventMap-rajapintaa indeksoidaan geneerisellä muuttujalla K, saadaan rajapinnasta muuttujan tyyppiä vastaava arvo, eli takaisinkutsufunktiossa oletetun datan tyyppi. Geneerinen tyyppi K saa tyyppiarvonsa funktion kutsujan ensimmäisestä syöteparametrista, eli tämän avulla funktio saa halutun tyypityksen antamalla odotettavan (tai kutsuttavan, mikäli tapahtuma kutsutaan emit-funktiolla) tapahtumatyypin arvon ensimmäiseksi parametriksi.

Itse varsinainen luokan tapahtumien kuuntelu ja odottaminen tapahtuu luokan ulkopuolella luodun olion avulla. Oliolla kutsutaan on-funktio, jolle annetaan ensimmäisenä parametrina odotettava tapahtuma ja toisena joko viittaus toisaalla ennalta määritettyyn takaisinkutsufunktioon tai itse takaisinkutsufunktio suoraan kirjoitettuna on-funktiokutsun toiseen parametriin. Tässä vaiheessa kehitysympäristö osaa määrittää, onko takaisinkutsufunktiolla syöteparametria vai ei ja mikäli on, myös sen tyypityksen. Tämä (kuin myös kaikki muukin edellä mainittu automaattinen tyypitys ja täyttö) edellyttää kuitenkin, että kehitysympäristö tukee Typescript-ohjelmointikielen reaaliaikaista tulkitsemista kielipalvelimen

avulla. Esimerkiksi VSCode-ohjelmointiympäristössä tämä on helppo lisätä asentamalla Microsoftin virallinen Typescript-laajennus.

8 Pohdinta ja omat havainnot

Opinnäytetyöprosessi alkaa kokonaisuudessaan olla loppusuoralla. Yleisesti ottaen koen työn onnistuneeksi, koska työssä päästään sille asetettuihin tavoitteisiin. Tässä osiossa koostan perustellusti omaa pohdintaani niin koko opinnäytetyöstä, kuin myös sen eri vaiheista, etenkin itse varsinaisesta ohjelmiston kehittämisprosessista ja tuon myös omia havaintojani esille. Lisäksi tuon esille myös eettisiä näkökulmia niin tehdystä työstä, kuin myös työn tuloksista. Tämä on siis eräänlainen pohtiva yhteenveto koko opinnäytetyön tekoprosessista.

Tyypillisesti opinnäytetyö alkoi aiheen selkeydyttyä suunnitteluprosessista, jossa luodaan pohjaa tulevalle opinnäytetyölle ja sen eri vaiheille. Suunnitteluprosessi kannatta tehdä perusteellisesti ja siinä tulee ajatella realistisesti. Optimistinen ja paljon asioita ylenkatsova suunnitelma voi olla hyvin hankala seurata. Vastapainona myöskään liian pikkutarkkaan suunnitteluun ei kannata mennä, koska muuten suunnitteluvaiheeseen kuluu todella paljon aikaa ja suunnitelmaa joutuu mukauttamaan heti ensimmäisen odottamattoman tilanteen tullessa. Omalla kohdallani tein näistä ensimmäisen virheen, joka oli optimistisuus aikataulua suunnitellessa, tästä lisää seuraavassa kappaleessa.

Merkittävin muutos alkuperäiseen suunnitelmaan nähden tapahtui opinnäytetyön aikataulutuksessa. Alkuperäisen aikataulun mukaan opinnäytetyön oli määrä valmistua joulukuuhun 2023, mutta työn matkan varrella ilmestyneiden ulkoisten hidasteiden vuoksi prosessi venyi kesäkuuhun 2024. Syynä tähän oli optimistinen aikataulutus ja aikatarpeet prosessin eri vaiheisiin. Ongelma olisi ratkennut ennakoimalla mahdolliset hidasteet paremmin ja yksinkertaisesti varaamalla enemmän aikaa kuhunkin työn eri vaiheeseen. Opinnäytetyön suunnittelu ja kirjallisuuskatsaus valmistuivat aikataulun mukaisesti kesällä 2023, mutta

itse kehitysprosessi etenkin projektin kehittämisen alkuvaiheessa venyi ja venyi, mikä puolestaan suoraan johti myös myöhempien prosessien aikataulun myöhästymiseen ja venymiseen.

Muilta osin opinnäytetyön suunnittelu onnistui hyvin. Ohjelmiston ja itse opinnäytetyön tavoitteet ja toteutustavat olivat selkeät alusta saakka ja suunnitelmassa on pysytty aina työn loppuun asti, poikkeuksena aiemmin mainittu ongelma aikataulutuksessa. Suunnitelma yleisesti oli kattava, mutta ei kuitenkaan liian pikkutarkka, vaan siinä asetettiin yleinen rajaus työn aiheeseen, listattiin käytettävät toteutustekniikat, työn prosessin eri vaiheet ja työn aikataulutus. Jos tekisin suunnitelman uudelleen tässä vaiheessa, pitäisin sen pääpiirteiltään samanlaisena, mutta aikataulutuksen laatisin kokonaan uudelleen.

Suunnitelman jälkeen työssä tulee tiedonkeruu. Tiedonkeruuprosessi eteni aikataulutuksen mukaan ja oli omasta mielestäni onnistunut. Tiedonkeruussa hyödynnetään paljon työssä käytettyjen standardien virallisia määrityksiä. Tämä on tietojen aitouden kannalta kriittistä. Eettisesti on tärkeää, että työn tietoperusta pohjautuu faktoihin ja että mahdollisimman vähän asioita jäisi oman arvelun varaan. Työn teknisyyden kannalta eduksi oli se, että tekniikat ovat pitkälti standardisoituja ja standardeja on hienosäädetty ja päivitetty joissain tapauksissa jopa kymmenien vuosien ajan, joten määritykset ovat kattavia ja tarkkoja. Yleisesti sanoisin tiedonkeruusta saadun tietopohjan olevan kattava ja edesauttaneen ohjelmiston kehitysprosessissa paljon.

Kun tietopohja on luotu, alkaa itse varsinainen työ. Tässä työssä se tarkoitti ohjelmiston kehittämisprosessia. Kehittämisprosessi on mielestäni koko työn huipentuma, koska siinä niin suunnitelmassa määritellyt asiat kuin myös tiedonkeruussa luotu tietopohja menevät käytäntöön. Prosessissa tulee pysyä määritellyissä toimintamalleissa ja siinä tulee hyödyntää tiedonkeruusta saatua tietoa ja kaiken tämän ohella pysyä aikataulussa. Työn teko onnistui hyvin koodin laadun ja toiminnallisuuden osilta. Testaamista olisi voinut tehdä laajemmin etenkin rajatapauksien osalta, sekä esimerkiksi IPv4-pakettien otsakkeiden options-kenttä voisi myös olla purettu auki, vaikkakin kenttää käytetään harvoin ja sen data ei

ole kovin olennaista. Toki tämä data kyllä puretaan ulos, mutta vain heksadesimaalipohjaisena merkkijonona. Tärkein asia toteutetussa koodissa oli sen toimintavarmuus ja suorituskyky. Koska ohjelmistolla on pääsy kaikkeen sille annetun verkkoportin datavirtaan, on tärkeää, että tämä tieto ei voi päätyä ulkopuolisille ja että koodissa ei ole tietoturvaa vaarantavia puutteita. Vaikka lisenssi vapauttaakin kehittäjän ohjelmiston käyttövastuusta, on silti tärkeää, että ohjelmiston käyttö on turvallista.

Versionhallintaa olisi myös voitu käyttää kattavammin etenkin haaroja käyttäen ja useampia solmuja toteuttaen, mutta tämä ei kuitenkaan kehitysprosessissa aiheuttanut hidasteita, eikä ongelmia. Pääsyynä tähän oli se, että ohjelmiston kehitysprosessi tapahtui yksin. Jos projektissa olisi ollut mukana useampi henkilö, olisi tämä ollut merkittävästi tärkeämpää. Merkittävin puute ja ongelmatekijä tässä projektin vaiheessa oli aikataulutus. Aiemmin määritetyssä aikataulussa ei pysytty ja projektin kehittäminen viivästyi huomattavasti suunnitelmaa pidemmälle. Kaikesta huolimatta ohjelmisto valmistui ja lopputulos vastaa hyvin ohjelmiston toiminnalle asetettuja tavoitteita.

Yleisesti ottaen opinnäytetyö ja siinä toteutettu ohjelmistoprojekti ovat onnistuneita. Projekti opetti paljon uutta tietoa ja tuntemusta verkkoliikenteen rakenteesta, sekä sen käsittelystä, niin ohjelmistossa kuin yleisesti. Projektin lopputulos vastaa sille asetettuja tavoitteita ja tarjoaa hyvät mahdollisuudet kenelle tahansa jatkokehitykseen. Koska kyseessä on kuitenkin hyvin geneerinen pohja, on ohjelmistosta kuitenkin valitettavasti myös mahdollista toteuttaa helposti pahantahtoisia kokonaisuuksia, jotka voidaan esimerkiksi jonkin haavoittuvuuden kautta ujuttaa kolmannen osapuolen järjestelmään luvanvastaisesti. Tätä toivottavasti ei tule tapahtumaan, mutta mahdollisuus siihen on, eikä sitä voi myöskään juuri mitenkään estääkään. Tämä tekee ohjelmistoon eettisyyden kannalta oman varjopuolensa, mutta toisaalta sama asia pätee lähes mihin vain avoimen lähdekoodin ohjelmistoihin, jotka keskittyvät tavalla tai toisella tietoverkkojärjestelmiin ja viestintään, joten en pidä sitä suurena uhkana tai haittapuolena. Olen kuitenkin tietoinen uhan mahdollisuudesta.

Projektin alkuperäinen lähdekoodi on löydettävissä GitHub-palvelusta (https://github.com/N1kO23/ONT/tree/ONT), haarassa ’ONT’. Kuka vain ohjelmistosta kiinnostunut on täysin tervetullut käyttämään lähdekoodia mieleisellään tavalla lisenssin rajoissa, laittomuuksia en kuitenkaan halua kenenkään tekevän. Ohjelmisto rakentuu Docker-kontiksi ja kehitysprosessissa ohjelmiston kontitus oli yleisesti todella helppoa, merkittävä etu tähän oli työelämästä saatu käyttökokemus Dockeriin, sekä valmiiksi olemassa olevan NodeJS-kontin käyttäminen kontin pohjana tarkoitti, että rakennusprosessi oli hyvin yksinkertainen.

Lähteet

Arkko, J & Pignataro, C. 2009. IANA Allocation Guidelines for the Address Resolution Protocol (ARP). https://www.rfc-editor.org/rfc/pdfrfc/rfc5494.txt.pdf. 02.05.2024.

Babak, B & Harrison, J & Mohammed, A. 2017. An Introduction to Docker and Analysis of its Performance. https://www.researchgate.net/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance. 02.08.2023.

Cecil, A. 2023. A Summary of Network Traffic Monitoring and Analysis Techniques. https://www.cse.wustl.edu/~jain/cse567-06/ftp/net_monitoring.pdf. 26.07.2023.

Dellinger, AJ & Aditham, K. 2023. What is PCAP? Packet Capture Explained. https://www.forbes.com/advisor/business/software/what-is-pcap/. 25.9.2023.

Deri, L. 2023. Improving Passive Packet Capture: Beyond Device Polling. http://luca.ntop.org/Ring.pdf. 12.06.2023.

Docker. 2023. Networking overview. https://docs.docker.com/network/. 18.08.2023.

Docker. 2024. What is a Container? https://www.docker.com/resources/what-container/#/package_software. 18.05.2024.

Eddy, W. 2022. Transmission Control Protocol (TCP). https://www.rfc-editor.org/rfc/rfc9293.pdf. 03.05.2024.

Fielding, R & Nottingham, M & Reschke, J. 2022. HTTP Semantics. https://www.rfc-editor.org/rfc/rfc9110.html. 14.08.2023.

Hirsikangas, S. 2023. Riverbed AirPcap Nx : Wi-fi-verkon liikenteen seuranta AirPcap-ohjelmistolla. https://www.theseus.fi/handle/10024/798998. 18.07.2023.

IEEE. 2003. IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks. New York, Yhdysvallat. The Institute of Electrical and Electronics Engineers, Inc.

Impreva. 2023. OSI Model. https://www.imperva.com/learn/application-security/osi-model/. 26.07.2023.

ISO/IEC 7498-1. 1994. Information Technology - Open System Interconnection - OSI Reference Model: Part 1 - Basic Reference Model. Sveitsi. ISO/IEC.

ITU. 1996. X.225 : Information technology - Open Systems Interconnection - Connection-oriented Session protocol: Protocol specification. https://www.itu.int/rec/T-REC-X.225-199511-I/en. 17.08.2023.

Juell, K. 2022. How To Build a Node.js Application with Docker. https://www.digitalocean.com/community/tutorials/how-to-build-a-node-js-application-with-docker. 16.08.2023.

Postel, J & Reynolds, J. 1985. FILE TRANSFER PROTOCOL (FTP). https://www.rfc-editor.org/rfc/pdfrfc/rfc959.txt.pdf. 06.05.2024.

Ruoho, S. 2019. Verkkoliikenteen valvonta ja analysointi. https://www.theseus.fi/handle/10024/168606. 12.08.2023.

Shaw, K. 2018. The OSI model explained and how to easily remember its 7 layers. https://www.networkworld.com/article/3239677/the-osi-model-explained-and-how-to-easily-remember-its-7-layers.html. 15.07.2023.

Tuomisto, S. 2019. Implementation of High-Performance Network Interface Monitoring. https://www.theseus.fi/handle/10024/172820. 17.07.2023.

Wesley, E. 2022. Transmission Control Protocol (TCP). https://datatracker.ietf.org/doc/html/rfc9293. 11.06.2023.