Antton Jokinen

Aaltomuoto-ohjelmiston arkkitehtuurin suunnitteleminen yhteensopivaksi OMNeT++-simulaatioympäristöön sekä laiteympäristöön

Insinööri (AMK)

Tieto- ja viestintätekniikan koulutus

Kevät 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Jokinen Antton

Työn nimi: Aaltomuoto-ohjelmiston arkkitehtuurin suunnittelu yhteensopivaksi OMNeT++-simulaatioympäristöön sekä laiteympäristöön

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: ohjelmistoarkkitehtuuri, ohjelmistosuunnittelu, simulointi, ohjelmistokehitys, tietokoneohjelmat

Simulaatiota hyödynnetään ohjelmistokehityksessä vuosi vuodelta yhä enemmän. Simulointikeinoja haetaan alan tekijöiden toimesta jatkuvasti. Simulaatioympäristössä ohjelmiston toiminnalle kuitenkin esitellään aivan uudenlaisia haasteita. Haluttaisiin, että aikaa ja resursseja ei käytetä ohjelmiston uudelleenkirjoittamiseen pelkästään simuloinnin vuoksi.

Tässä työssä käsiteltiin aaltomuoto-ohjelmiston ohjelmistoarkkitehtuurin yhteensopivuutta OMNeT++-simulaatioympäristöön. OMNeT++ on avoimen lähdekoodin simulaatioympäristö, joka on suunniteltu verkostojen ja tiedonsiirron simulointia varten. Aaltomuoto-ohjelmisto on ohjelmistokokonaisuus, joka on tyypillisesti suunniteltu tiedonsiirtoon ohjelmistoradioalustalle. Simuloitavan aaltomuoto-ohjelmiston suunnittelussa ei ollut huomioitu simulointia, joten epäyhteensopivuudet olivat oletettuja. Löydetyt epäyhteensopivuudet käsiteltiin tarkasti ja niitten pohjalla olevat syyt tarkasteltiin. Ratkaisut epäyhteensopivuuksiin halutaan ottaa huomioon ja soveltaa ohjelmiston arkkitehtuuriin jo suunnitteluvaiheessa.

Aaltomuoto-ohjelmiston tuominen simulaatioympäristöön estyi ohjelmistoarkkitehtuurillisten epäyhteensopivuuksien aiheuttamien ongelmien vuoksi. Aaltomuodon C++-ohjelmistokomponenttien rakenteen vuoksi, joiden toiminnassa hyödynnettiin singleton-menetelmää, ohjelmistojen yhteensopivuus todettiin mahdottomaksi ilman refaktorointia. Simulaatioympäristössä ajettavat ohjelmistot pakotetaan olemaan ajossa yksittäisessä prosessissa, jonka avulla simulaation toiminta mahdollistetaan. Täten samalla kuitenkin estetään aaltomuodon toiminta simulaatiossa, sillä aaltomuoto-ohjelmiston ohjelmistoarkkitehtuuri ei ole suunniteltu ajettavaksi yhdessä prosessissa.

Yksi työssä ehdotetuista ohjelmistoarkkitehtuurillisista rakenteista on konttirakenne, jossa ohjelmiston ydintoiminnalta abstraktoidaan yhteydet muille ohjelmistokomponenteille ympäröimällä ohjelmiston toiminta kontilla. Kontti muodostuu yhteysrajapinnoista ohjelmiston ulkopuolelle. Konttiarkkitehtuuria soveltamalla suunnitteluvaiheessa ohjelmiston toiminta voidaan pitää irrallisena ohjelmiston ajoympäristöstä, jolloin ohjelmiston yhteensopivuutta erilaisiin ajoympäristöihin, kuten simulaatioympäristöön, saadaan parannettua.

Työssä todettiin, että ohjelman simulointiin vaikuttaa laaja määrä ohjelmistoarkkitehtuurillisia tekijöitä. OMNeT++:n simulaatiokernelin toiminta asettaa useita pakotteita simuloitavalle ohjelmistolle, joita on hankala kiertää silloin, kun ohjelmisto on jo toteutettu. Vaikuttavimmat rajoitteet olivat singleton-menetelmän hyödyntäminen ja verkkopistokerajapintojen käyttäminen ohjelmistossa. Täten, löydettyjä ongelmallisia ohjelmistoarkkitehtuurillisia päätöksiä on vältettävä jo ohjelmiston suunnitteluvaiheessa, jotta ylimääräiseltä työltä säästytään.

KAMK • University of Applied Sciences

Abstract

Author(s): Jokinen Antton

Title of the Publication: Designing Waveform Software Architecture to Be Compatible with OMNeT++ Simulation Environment and Hardware

Degree Title: Bachelor of Engineering, Information and Communications Technology

Keywords: software architecture, software design, simulating, software development, computer programs

Simulating is used for software development and testing more and more every year. There is a constant ongoing search for new innovative ways to benefit from simulation in software development. However, a simulation environment introduces several new types of challenges for the simulated software, especially if the software is originally designed to be operated on a structurally different platform. The end goal is to use as few resources as possible but achieve compatibility for the simulated software and simulation environment so that simulation results can be obtained.

In this thesis, the compatibility between a waveform software and the OMNeT++ simulation environment was researched. The waveform software has not been designed with simulating in mind, so factors affecting compatibility were to be expected. These factors were examined carefully and the root causes behind them investigated. If possible, the problematic factors should be avoided in the future by designing the software architecture to take them into account. Software architecture models preventing the discovered problematic factors were developed to alleviate the problems.

Importing the waveform software into OMNeT++ was prevented when problematic design patterns used in the C++ code of the waveform software could not be avoided. The single most problematic design pattern used was the dreaded singleton-pattern, which effectively rendered the software impossible to simulate without major structural refactoring. Software running in simulation environment is forced to run in a single process, which is necessary due to the structure of the OMNeT++ simulation kernel. As the waveform software was designed to be used exclusively in hardware, the used design patterns caused compatibility issues when used in the simulation environment.

The discovered problematic factors affecting compatibility should be avoided during the design phase of the software architecture. This saves the developers from having to do additional work to be able to simulate the software effectively. One of the suggested architectural design patterns is the container architecture, in which the connections of the software to external components should be abstracted by surrounding the core functionality of the software with a container. The container is formed of interface classes connecting the software to external components.

In the end, the factors affecting compatibility between the waveform software and the simulation environment were too sizeable to overcome without any refactoring. The functionality of the OMNeT++ simulation kernel sets restraints for the software to be simulated that are difficult to overcome when the functionality of the software has already been implemented. These restraining factors include the use of the singleton design pattern and network socket interfaces connecting software components. The problematic software architecture factors should be avoided during the design phase so that no additional work needs to be done later.

Sisällys

Lyhenteet ja määritelmät

@directin	OMNeT++:n porteille määriteltävä decorator (ominaisuus), joka antaa portille ominaisuuksia. Tämä määritelmä antaa portille kyvyn vastaanottaa viestejä vaikkei portti olisi yhdistetty toiseen porttiin.
AIF	Air Interface. Ilmarajapinta. Ohjelmistoradion tapauksessa ilmarajapinta mahdollistaa ohjelmistolle datan lähettämisen ja vastaanottamisen radiotaajuuksia hyödyntäen.
API	Application Programming Interface. Ohjelmointirajapinta. Tarjoaa ohjelmoijalle tavan hyödyntää jo olemassa olevia ratkaisuja ongelmiin, yksinkertaistaen lopullista ohjelmaa parantaen sen luettavuutta ja ymmärrettävyyttä.
CSMA	Carrier Sense Multiple Access. MAC-kerroksen protokolla, joka varmistaa kaistan vapauden ennen datan lähettämistä.
DES	Discrete Event Simulator/Simulation. Simulaatio-ohjelmiston toimintatapa, jossa simulaation eteneminen pohjautuu tapahtumien vastaanottamiseen ja toteuttamiseen.
INI	Initialization file. OMNeT++:n tarjoama tiedostotyyppi, jonka avulla voidaan määritellä erilaisia parametreja simulaatiomallille. [1.]
MAC	Medium Access Control. OSI-mallin kerros, joka hallitsee fyysisiä komponentteja, jotka ovat vastuussa lähettämisestä.
NED	Network Description Language. OMNeT++:n tarjoama ohjelmointikieli, jonka avulla voidaan määritellä moduuleja simulaatiomallia varten. [2.]
OMNeT+	Objective Modular Network Testbed in C++. Opetus- ja tutkimuskäyttöön ilmaiseksi saatavilla oleva avoimen lähdekoodin simulaatio-ohjelmisto, jota pääsääntöisesti hyödynnetään verkkojen toiminnan simulointiin.
OSI-malli	Open Systems Interconnection model. Konseptipohjainen malli, joka jakaa kommunikointiin tarkoitettujen ohjelmistojen vastuut ohjelmistokerroksiin.

SDR	Software Defined Radio. Ohjelmistoradio, jonka toimintaa voidaan muokata ohjelmiston avulla [3].
Simulaatiokerneli	Simulaatio-ohjelmiston ydintoiminnallisuus, joka ohjaa simulaatiota, simulaation komponentteja ja moduuleja.
Skenaario	Simulaatioympäristössä luotu virtuaalinen tilanne, jossa simuloitavan kohteen toimintaa halutaan testata.

1 Johdanto

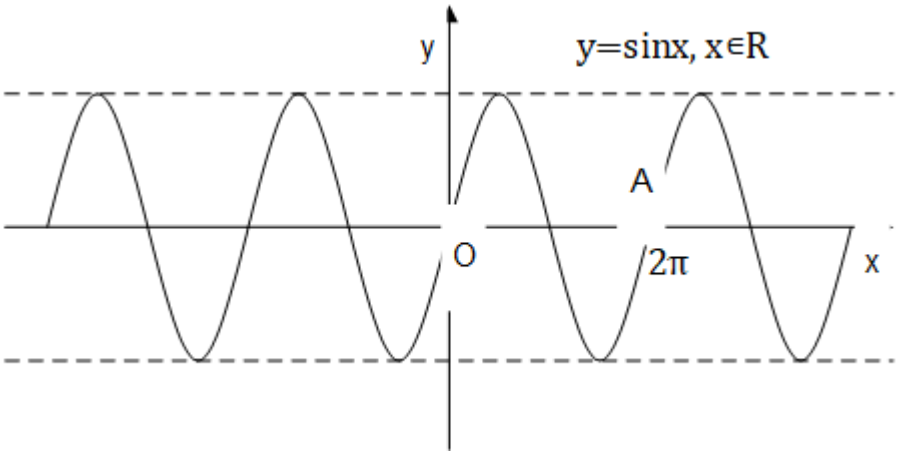
Tämä opinnäytetyö on tuotettu toimeksiantona Bittium Wireless -yritykselle Kajaanissa. Toimeksiantajan asettama tavoite työlle oli selvittää, minkälaiset tekijät ja suunnittelupäätökset ohjelmistoarkkitehtuurissa vaikuttaisivat aaltomuoto-ohjelmiston yhteensopivuuteen simulaatioympäristöön. Kun nämä tekijät ovat jo ohjelmiston suunnittelun aikana tiedossa, mahdollistuu niiden huomioiminen ohjelmistoarkkitehtuurin rakenteen kehityksen aikana, seuraten *Simulation Driven Development* (Simulaation ohjaama kehitys) -nimisen ohjelmistokehitysprosessin asettamia askelmerkkejä.

Simulaatio lukeutuu yleisen ohjelmistokehityksen määritelmään. Sen hyödyntäminen kehityksen aikana auttaa tunnistamaan mahdollisia vikatilanteita ja testaamaan tavallisin keinoin hankalasti toistettavia tilanteita. Simulointia halutaan kehitysprosessissa lisätä, mutta sen mukanaan tuomia resurssipakotteita vähentää.

Tässä dokumentissa käsitellään Bittiumin kehittämän ohjelmistoradiolle suunnatun aaltomuoto-ohjelmiston tuomista OMNeT++-simulaatioympäristöön ja sen aikana todettuja ohjelmistoarkkitehtuurillisia kohtia, jotka vaikuttivat sovittamisprosessiin. Sen lisäksi käsitellään, miten ongelmallisia kohtia voitaisiin vältellä ja sopivuutta simulaatioympäristöön parantaa.

2 Aaltomuodot

Aaltomuoto (waveform) on termi, jota käytetään monella teknologian alalla. Yleisesti kuvailtuna aaltomuodolla tarkoitetaan signaalin muotoa ajan funktiona [4]. Aaltomuotoja hyödynnetään elektroniikan alalla vaihtelevien jännitteiden tai virtojen havainnollistamiseen. Akustiikassa aaltomuodolla kuvaillaan tyypillisesti ääntä visuaalisessa muodossa. Kuvassa 1 voidaan nähdä siniaalto havainnollistettuna aaltomuotona.



Kuva 1. Siniaalto aaltomuotona.

Tässä tekstissä termin aaltomuoto käyttö referoi yksinomaisesti aaltomuoto-ohjelmistoon tai aaltomuoto-ohjelmiston protokollakerroksiin. Luvussa aiemmin esitelty aaltomuodon määritelmä, jota käytetään virtojen ja jännitteiden havainnollistamiseen, ei lukeudu tämän opinnäytetyön aihealueeseen.

Aaltomuoto-ohjelmisto saattaa kuulostaa terminä samankaltaiselta aaltomuoto-termiin verrattuna, mutta termeissä on laaja merkityksellinen ero. Aaltomuoto-ohjelmisto on ohjelmistokokonaisuus, joka on suunnattu käytettäväksi ohjelmistoradioalustalle. Ohjelmistoradio (software defined radio, SDR) on radio, jonka toiminta perustuu korkealla tasolla fyysisten radiokomponenttien korvaamiseen ohjelmistokomponenteilla. [5.] Ohjelmistoradiota hyödynnettäessä alustana radion toimintaa on mahdollista muuttaa laajasti ohjelmiston avulla. Täten koko radion toimintaa on helpompi muokata.

Ohjelmistoradioiden teoria esiteltiin ensimmäisen kerran 1990-luvulla. Nykyään ohjelmistoradioita käytetään erilaisiin hyötytarkoituksiin ympäri maailmaa. Terveydenhuolto, astronomia, satelliittien hyötydata tai opetuskäyttö ovat vain muutamia käyttötarkoituksia ohjelmistoradioille

[6.] Ohjelmistoradioita voi myös hyödyntää tavallisten radioiden käyttötarkoituksiin, kuten puheen tai datan välittämiseen. Työssä myöhemmin mainitut ohjelmistoradiot ovat suunniteltu puheen ja datan välittämiseen.

Aaltomuoto-ohjelmisto koostuu kaikista aaltomuodon toimintaan tarvittavista ohjelmistokomponenteista. Tässä työssä simuloinnin kohteena olivat vain aaltomuoto-ohjelmiston protokollakerrokset (waveform software protocol layers), joten kaikki protokollakerrosten ulkopuoliset ohjelmistokomponentit jätettiin huomiotta. Aaltomuoto-ohjelmiston protokollakerrokset ovat vastuussa ohjelmistoradion aaltomuoto-ohjelmiston suunnitellusta toiminnallisuudesta.

Aaltomuoto-ohjelmiston tyypillinen toiminta ohjelmistoradioalustalla on puheen ja/tai datan siirto. Näitä ohjelmiston pääominaisuuksia halutaan simuloida myös OMNeT++-simulaatioympäristössä. Kyseisen kaltaisella aaltomuoto-ohjelmistolla ohjelmistoradion toiminta muistuttaa pintapuolisesti perinteisen radion toimintaa. Kuten aikaisemmin kuitenkin todettiin, ohjelmistoradio tarjoaa laajempaa monimuotoisuutta ja muokkautuvuutta verrattuna perinteisiin radioihin ohjelmistopohjautuvuutensa ansiosta.

3 Simulointi

Termillä simulointi voi olla useampi tarkoitus. Se voi tarkoittaa kenties erilaisten nesteiden ominaisuuksien toteamista, erilaisten liikenneristeysten tehokkuuden tarkistelua tai vaikkapa videopeliä, joka koettaa matkia todellisuutta. Perinteisesti simulaatiot on jaettu kahteen eri luokkaan: analyyttisiin simulaatioihin sekä virtuaalisen ympäristön simulaatioihin. Analyyttisen simulaation tarkoitus on saada selville jonkin systeemin toiminta mahdollisimman tarkkaa alkudataa hyödyntäen. Tämän vuoksi analyyttisen simulaation ajaminen tulisi olla mahdollisimman nopeata sekä tuottaa aina sama lopputulos olettaen, että alkuarvot ja muut muuttujat eivät vaihdu simulaatiokertojen välissä. [7, s. 6.] Tässä opinnäytetyössä simuloinnilla tarkoitetaan aaltomuoto-ohjelmiston protokollakerrosten toiminnan simulointia, joka voidaan lukea analyyttiseksi simuloinniksi. Simuloinnin käsittely yleisesti rajataan ohjelmistosimuloinnin näkökulmaan.

Opinnäytetyötä varten simulaattoriksi toimeksiantaja oli valinnut OMNeT++-simulaatioympäristön. OMNeT++ on modulaariseen rakenteeseen perustuva simulaatioympäristö, jonka on kehittänyt OpenSim Ltd [8]. OMNeT++:n ensimmäinen versio on julkaistu jo viime vuosituhannen puolella. Ensimmäinen viittaus OMNeT++:n versiodokumentissa on päivätty syyskuussa 1997, kun taas ensimmäinen julkaistu versio OMNeT++:n GitHub-sivuilla on päivätty syyskuuhun 1999 [9]. OMNeT++:n toimintaa käsitellään tarkemmin myöhemmin dokumentissa.

Simuloinnin kohde työssä oli aaltomuoto-ohjelmiston protokollakerrosten toiminnan simulointi. Ohjelmisto on toteutettu C++-ohjelmointikielellä, joka on yhteensopiva OMNeT++-simulaatioympäristön kanssa. Simulaation tavoitteeksi asetettiin toimeksiantajan puolesta aaltomuoto-ohjelmiston CSMA-algoritmien simulointi. Itse opinnäytetyön tavoite oli kuitenkin löytää aaltomuodosta sekä simulaatioympäristöstä ohjelmistoarkkitehtuurillisia epäyhteensopivuuksia, jotka tulisi ottaa huomioon aaltomuoto-ohjelmiston suunnittelussa ja kehityksessä. Toimeksiantaja haluaa olla tietoinen OMNeT++-simulaatioympäristön ohjelmistoarkkitehtuurin rakenteen asettamista vaatimuksista simuloitavalle ohjelmistolle, jotta mahdollisia epäyhteensopivuuksia voitaisiin vältellä jo simuloitavan ohjelmiston suunnitteluvaiheessa.

3.1 Simuloinnin tavoite

Simuloinnilla on tyypillisesti jokin tavoite, joka halutaan saavuttaa. Kenties simuloitavasta ohjelmistosta halutaan dataa, joka osoittaa ohjelmiston toiminnallisuuden tai toimimattomuuden. [7,

s. 9.] Tietoliikenneverkoissa simulointi on usein hyödynnetty keino, sillä se saattaa olla ainoa realistinen keino toteuttaa monimutkaisia tai epätavallisia olosuhteita verkolle. Oletetaan, että verkko-ohjelmiston toimintaa halutaan testata niin, että verkko muodostuu 100 000 laitteesta, jotka viestivät keskenään. Kyseinen tilanne ei ole välttämättä kovin realistinen. Tästä huolimatta ohjelmiston testaaminen äärimmäisissä olosuhteissa voi paljastaa vikoja, joita ei olisi mahdollista löytää muuten. 100 000 laitteen verkkoa ei ole helppo testata käsin, koska testin pelkät laitevaatimukset ovat valtavat, puhumattakaan henkilömäärästä, joka tarvittaisiin niiden käyttämiseksi. Simulaatiossa 100 000 laitteen simulaatio kuitenkin onnistuisi, joten se on ainoa realistinen vaihtoehto. On kuitenkin huomioitava, että jos ohjelmisto vaatii muutoksia toimiakseen simulaatiossa tai vain tiettyä osaa ohjelmistosta simuloidaan, on mahdollista, että tietyt viat eivät ilmene simulaatioympäristössä, jotka ilmenisivät muissa ympäristöissä.

Simuloinnin avulla saavutettava tavoite vaihtelee simuloitavan toiminnan mukaisesti. Tässä työssä tavoitteena oli kartoittaa mahdollisimman laajasti, minkälaisia ohjelmistoarkkitehtuurillisia tekijöitä tulee aaltomuoto-ohjelmiston suunnittelussa ottaa huomioon, kun tarkoituksena on simuloida sen toimintaa.

Perinteinen tapa testata ohjelmiston toimintaa ja vahvistaa ohjelmistolle asetettuja vaatimuksia simuloinnin avulla on toteuttaa ohjelmiston osat, joiden toiminta halutaan vahvistaa simuloinnin avulla, uudestaan simulaatioympäristössä. Toimintatapana sillä on omat puutteensa, sillä ohjelmiston ominaisuuksien toteuttaminen uudestaan simulaatioympäristössä vaatii resursseja. Yksi tämän tutkimuksen päämotivaatioista olikin, onko näitä resurssitarpeita mahdollista vähentää tai jopa välttää kokonaan.

Opinnäytetyön tavoitteena oli tutkia, onko jo toteutettua ohjelmistoa mahdollista simuloida. Valmiin ohjelmiston simulointi helpottaisi ohjelmiston jatkokehitystä, kuten uusien versioiden tuottamista. Simuloinnin tavoitteena oli tämän opinnäytetyön puitteissa aaltomuoto-ohjelmiston CSMA-algoritmien simulointi. Simulaatiosta haluttiin tuloksena dataa, joka osoittaisi mahdollisia virheitä algoritmien toteutuksessa tai vaihtoehtoisesti varmistaisi niiden toiminnan. Simulaation avulla on mahdollista toteuttaa haastavia sekä epätavallisia olosuhteita ohjelmistolle, jotta voidaan varmistaa sen toimintaa mahdollisimman monipuolisesti.

Tyypillisessä simulaatioympäristössä on mahdollista kerätä valtava määrä dataa simulaation ajalta. On käyttäjän vastuulla määritellä, minkälaista dataa halutaan kerätä ja kuinka paljon. Simulaatiosta saatua dataa halutaan luonnollisesti hyödyntää mahdollisimman laajasti, joten käyttäjän tulee miettiä tarkasti, mikä data on merkityksellistä kyseisessä simulaatiossa.

Simulaatiota toteuttamalla saavutettiin itse työn tavoite. Dokumentissa käydään laajamittaisesti läpi erilaisia ohjelmistoarkkitehtuurillisia tekijöitä, jotka vaikuttavat ohjelmiston kykeneväisyyteen toimia OMNeT++-simulaatioympäristössä, vaikkei niitä olisi alun perin suunniteltu siihen.

3.2 Simulointityökalut

Simulointityökaluja on vastaavasti monenlaisia [10]. Tässä dokumentissa käsitellään vain OMNeT++-simulaatioympäristöä, joten sitä kutsutaan dokumentissa simulaatioympäristöksi. Simulaatioympäristöjen ensisijainen tavoite on tarjota käyttäjälle ympäristö, joka ohjelmistojen tapauksessa mahdollistaa ohjelmiston laajamittaisen testauksen erilaisissa tilanteissa ja olosuhteissa. Erilaisia simuloituja tilanteita käyttäjä voi luoda käyttäen simulaatioympäristön tarjoamia työkaluja. Verkkoja simuloitaessa on yleistä, että halutaan testata erilaisia verkkomuodostelmia ja rakenteita. Täten voidaan löytää ongelmia laajamittaisista tai epätavallisista olosuhteista, mitä ei kyettäisi löytämään fyysisillä laitteilla testatessa yhtä helposti ja nopeasti.

Kaikenlainen simulointi perustuu simulaatiomalleihin. Simulaatiomalli muodostetaan OMNeT++:n tapauksessa luomalla periytyneitä C++-luokkia simulaatiokirjaston tarjoamista C++-luokista. Näihin periytettyihin C++-luokkiin käyttäjän tulee toteuttaa testattavan ohjelmiston toiminnallisuus, jota halutaan testata tai jonka toiminta halutaan vahvistaa. Työssä otettiin selvää, onko mahdollista liittää valmiin ohjelmiston toiminta näihin periytettyihin luokkiin, jotta ohjelmiston toimintaa voitaisiin simuloida. Kun periytettyjä C++-luokkia liitetään NED-kielellä määriteltyihin moduuleihin, simulaatiokerneli tietää ottaa vastuulleen niiden tilan päivittämisen simulaation aikana. [11, s. 32–100.]

OMNeT++:n ohjelmistoarkkitehtuurin toimintaa voi verrata pelimoottoriin. Molemmat tarjoavat käyttäjälle mahdollisuuden toteuttaa ohjelmoiden omaa toiminnallisuutta, joka tulee liittää simulaatiomallissa aktiivisena olevaan moduuliin, jotta sen toiminnallisuus toteutuu. Molempien olemassa olevien moduulien toimintaa, liikettä sekä aikaa hallitsevat simulaatioympäristön tai pelimoottorin sisäiset ohjelmistokomponentit, simulaation tapauksessa simulaatiokerneli. [12.]

Työssä hyödynnetty simulaatioympäristö on OMNeT++-versio 5.6.2 ja käyttöjärjestelmä on versioltaan Ubuntu 18.04 LTS.

3.3 OMNeT++/OMNEST

OMNeT++ on verkkoyhteyksien ja langattoman viestinnän simulointiin tarkoitettu simulaatioympäristö. Nimi lyhentämättömänä on *Objective Modular Network Testbed in C++* [13]. OMNeT++ on ilmainen akateemisiin käyttötarkoituksiin, mutta olemassa on myös kaupallinen versio. Kaupallinen versio on nimeltään OMNEST [14]. OMNEST ei ominaisuuksiltaan eroa OMNeT++:sta niin suuresti, ettei tietoa, joka on löydetty käyttämällä OMNeT++-ohjelmistoa voisi hyödyntää OMNEST-ohjelmistoon. Täten opinnäytetyön löydökset ovat hyödynnettävissä myös OMNEST-ohjelmiston käyttöä varten. [15.]

OMNeT++:n ohjelmistoarkkitehtuuri on suunniteltu pääsääntöisesti kommunikaatioverkostojen simulointi mielessä. Kuitenkin, sen sijaan, että OMNeT++:n käyttö olisi ollut ominaisuuksiltaan kohdennettu erityisesti kommunikaatioverkostojen simulointiin, OMNeT++:n toteutuksesta tehtiin tarkoituksella niin abstrakti kuin mahdollista. [16.] Moduulihierarkia, johon OMNeT++:n toiminta perustuu, ei edellytä ohjelmistolta tietynlaista rakennetta tai arkkitehtuuria. Yksittäinen moduuli voi sisältää rajattoman määrän toteutusta, vaikka tämä ei ole suositeltua. Päätös tehdä OMNeT++:n toteutuksesta mahdollisimman abstrakti on mahdollistanut OMNeT++:n käytön valtavaan määrään erilaisia simulaatiotavoitteita, ei pelkästään kommunikaatioverkostojen simulointiin.

3.3.1 INET

OMNeT++-ohjelmisto mahdollistaa monenlaisten verkkojen simuloinnin. Yksi suosituimmista työkaluista OMNeT++-käyttäjien keskuudessa on INET-framework (ohjelmistorunko). INET-ohjelmistorunko on avoimen lähdekoodin lisäkirjasto OMNeT++:n käyttöön. INET koostuu pääosin ohjelmistokomponenttipohjista, joista käyttäjät voivat muokata omiin simulaatiotarpeisiinsa sopivia komponentteja. [17.] Komponenttipohjista muokattujen versioiden luominen on toteutettu C++-periytymisen avulla. Kyseiset komponentit voivat lieventää simulaation toteutukseen vaadittua aikaa, sillä ne tarjoavat valmiita toteutuksia suurelle osalle tietoliikenneverkkojen simulointiin käytettävistä ominaisuuksista. Tämän vuoksi INET-ohjelmistorunkoa on hyödynnetty valtavasti OMNeT++-simulaatioympäristön käytössä. INET tarjoaa komponenttitoteutuksiensa lisäksi valtavasti simulaatioista opettavia dokumentteja ja ohjeita. [18.]

INET-ohjelmistorunkoa ylläpitävät ja päivittävät nykyään OMNeT++ tiimin jäsenet [17]. INET:n toteutus on tästä huolimatta avointa lähdekoodia ja kuka vain voi osallistua INET:n kehitykseen ja parantamiseen.

Tämän työn simulaatiotavoitteen puitteissa INET-ohjelmistorunkoa hyödynnettiin eri aaltomuoto-ohjelmiston komponenttien välisten yhteyksien luomiseen, johon hyödynnettiin INET:n tarjoamaa simulaatioympäristöön sopivaa *TCPSocket*-luokkaa (TCP-protokollaa hyödyntävä verkkopistoke). Tämän lisäksi INET:n tarjoamia *PropagationDelay*-malleja (etenemisviivemalli) hyödynnettiin simuloitujen radioiden yhteyksien simulointiin.

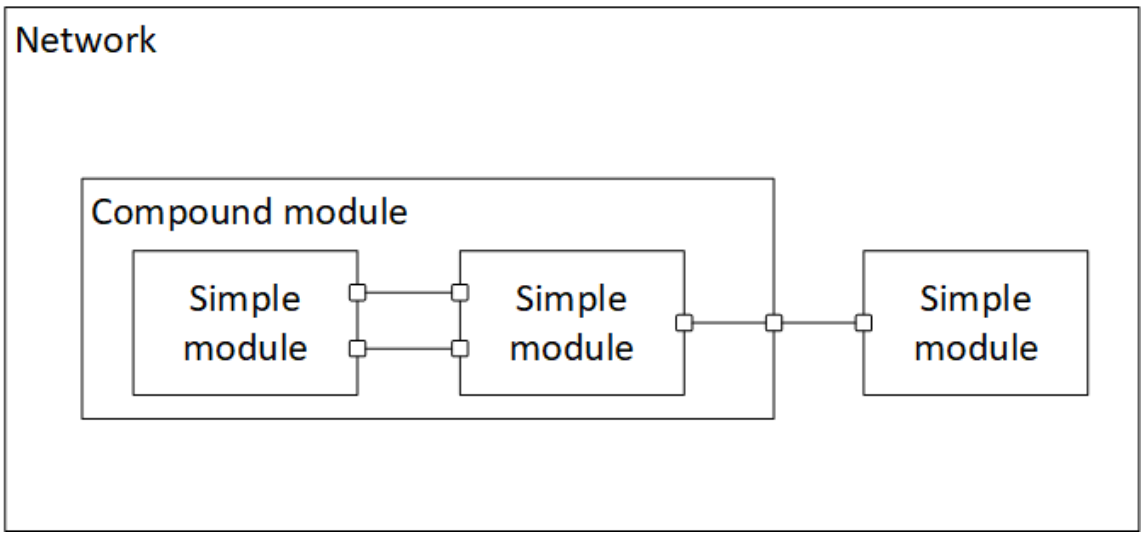
3.3.2 Simulaatiomalli & skenaario

Simulaatiomalli OMNeT++ -simulaatioympäristössä koostuu moduuleista. Moduuleja voisi rinnastaa tietynlaisiksi rakennuspalikoiksi, joista simulaation toiminnallisuus lopulta koostuu. Kaikki moduulit määritellään OMNeT++:n sisäisellä NED-ohjelmointikielellä (Network Description language, verkon kuvailukieli). Simulaatiomalli vastaa simuloitavan ohjelmiston toiminnallisuuden ohjaamisesta simulaatiokernelille. Simuloitavan ohjelmiston lähettäessä paketin verkkoliikennettä hyödyntäen simulaatiomallin tulee tietää, miten paketti saadaan määränpäähänsä. Simulaatiomalli toimii eräänlaisena rajapintana simuloitavan ohjelmiston ja simulaatiokernelin välillä, teeskennellen ohjelmistolle, että se on ajossa oikeassa ympäristössä, jotta ohjelmiston toimintaa voidaan tarkastella mahdollisimman realistisissa olosuhteissa.

Moduuleja on kahdenlaisia: yksinkertaisia moduuleja (simple modules) ja yhdistelmämoduuleja (compound modules). Yksinkertaisiin moduuleihin on käyttäjän mahdollista toteuttaa toiminnallisuutta C++-ohjelmointikielellä. Täten yksinkertaiset moduulit muodostavat simuloitavan ohjelmiston osia, tai jopa koko ohjelmiston. OMNeT++:n yleinen moduulihierarkia on näkyvillä kuvassa 2. [19.]

Yhdistelmämoduulit muodostuvat yhdestä tai useasta yksinkertaisesta moduulista. Yhdistelmämoduuleihin ei ole mahdollista toteuttaa toimintaa C++:n avulla, vaan ne koostuvat pelkästään yksinkertaisista moduuleista. Nämä periaatteet muodostavat OMNeT++:n tarjoaman moduulihierarkian. Moduulihierarkian tasot eivät ole määrältään rajoitettuja, vaan useampia yhdistelmämoduuleja voidaan yhdistää yhteen tai useampaan yhdistelmämoduuliin. Moduulihierarkian matalin taso rakentuu aina C++-koodista.

Kuvassa 2 on havainnollistettu OMNeT++:n moduulihierarkia korkealla tasolla. Yksi yhdistelmämoduuli sisältää kuvan tapauksessa kaksi yksinkertaista moduulia, jotka on yhdistetty toisiinsa kahdella portilla. Oikeanpuolisen yksinkertaisen moduulin kolmas portti on kytketty yhdistelmämoduulin porttiin, joka taas on kytketty yhdistelmämoduulin ulkopuolisen yksinkertaisen moduulin ainoaan porttiin. Kaikki kuvassa näkyvät moduulit sijaitsevat simulaatiossa verkkomoduulissa, joka kuvassa näkyy nimellä *Network*.



Kuva 2. OMNeT++:n yleinen moduulihierarkia.

Moduulit kommunikoivat keskenään viestimällä. Viestiminen tapahtuu kanavien ja porttien avulla tai vaihtoehtoisesti suoraan moduulilta toiselle. Moduuleille käyttäjä voi määritellä portteja, jotka tulee yhdistää haluamaansa porttiin. Portit on havainnollistettu laatikoina moduulien välissä. Moduulien portteja voidaan yhdistämisen jälkeen käyttää C++-koodissa viestien lähettämiseen toisille moduuleille. Jos yhteen tai useampaan yhdistelmämoduuliin kuuluva yksinkertainen moduuli haluaa lähettää viestin toiselle moduulille, kaikkien kahden moduulin välillä olevien porttien tulee olla yhdistetty. Viesti voi sisältää mitä tahansa dataa ohjelmisto on lähettämässä, eikä sen sisältö ole koon kannalta tai muuten rajoitettu. Porteille voidaan määritellä *decorator*eita (ei suoraa käännöstä, ominaisuuksia). @directin on decorator, joka antaa portille ominaisuuden vastaanottaa viestejä, vaikkei se olisi yhdistetty toiseen porttiin. Decoratoreita on OMNeT++:ssa saatavilla pitkä lista.

Simulaatiomallin toiminta voidaan jakaa kahteen alikategoriaan: mallin toiminnallisuuteen sekä mallin topologiaan. Kuten edellä kerrottiin, simulaatiomallin toiminnallisuus määritellään yksinkertaisen moduulien C++-luokissa. Mallin topologia sen sijaan määräytyy moduulien hierarkian mukaisesti. [20.]

OMNeT++ tarjoaa käyttäjälle mahdollisuuden määrittää erilaisia simulointiskenaarioita, joissa simulaatiomallia hyödynnetään. Useamman skenaarion määrittely ei ole pakollista, mutta niiden oikeaoppinen käyttö mahdollistaa laajempaa hyötyä simuloinnista. Mallin toiminnallisuuden luotettavuus korreloi lineaarisesti skenaarioiden määrään, jossa mallin toimintaa on testattu. Simulaatioskenaarioiden luomiseen käytetty aika on oletettavasti vähemmän kuin skenaarioiden testaaminen ilman simulaation hyödyntämistä, joten on suositeltavaa määritellä tarvittavat skenaariot ennen ohjelmiston toteutuksen tai simulaatiomallin toteutuksen aloitusta. Täten voidaan olla varmempia siitä, että simuloitavat skenaariot ovat mahdollisia toteuttaa.

OMNeT++-skenaarion määrittely koostuu simuloitavan verkon määrittelystä sekä simulaation moduulien tarvitsemien parametrien määrittelystä. Skenaarioiden erottelu tapahtuu INI-tiedostojen avulla. Yhtä skenaariota vastaa yksi INI-tiedosto. INI-tiedostoon käyttäjä voi määritellä moduulien tarvitsemia muuttujien arvoja. INI-tiedostot mahdollistavat eri tavalla määriteltyjen tilanteiden simuloinnin. Simuloitavalla skenaariolla tulee aina olla vähintään määritelty yksi verkko (network). Verkko on ainutlaatuinen moduuli OMNeT++:n silmissä, sillä se ei sisällä yhteyksiä muihin moduuleihin. Verkkomoduuli sisältää kaikki skenaariossa käytettävät moduulit.

OMNeT++ on modulaarinen simulaatioympäristö, mikä asettaa rajoituksia sen käytölle. Simulaation toiminta perustuu moduulien ympärille ja kaikki ajettava koodi tulee kuulua yhteen tai useampaan moduuliin, jotka ovat aktiivisena skenaariossa. OMNeT++:n sisäiset ydinkomponentit saavat hallita näitä moduuleja. Tämä tarkoittaa käytännössä sitä, että OMNeT++ ilmoittaa moduuleille simulaatioajan muutoksista, moduulin sijainnista skenaariossa, jos se on skenaariolle relevanttia sekä pakettien saapumisesta moduulille. Tätä periaatetta seuraten käyttäjän simuloitava toiminnallisuus tulisi toteuttaa moduuleihin, jotka ovat aktiivisena skenaariossa.

Kyseisen periaatteen takia OMNeT++:n käyttö simulointia varten vie paljon aikaa ja resursseja. OMNeT++:n ohjelmistoarkkitehtuuri ei mahdollista moduuleihin jakamattoman koodin käyttöä ilman suurta ylimääräistä vaivaa. Sen sijaan, ohjelmoijan oletetaan toteuttavan simuloitava ominaisuus ensin OMNeT++:n sisällä moduuleja käyttäen ja myöhemmin, kun ominaisuuden toiminta on vahvistettu, toteuttamaan sen uudestaan laiteympäristöön ilman moduuleja. Edellä kuvatut ongelmat ovat suurin motivaatio toimeksiantajan puolelta tämän ongelman edistämiseen, sillä tällä hetkellä simuloinnin käyttö vie suuren määrän resursseja.

3.4 Discrete Event Simulation

OMNeT++ on DES-periaatetta noudattava ohjelmisto. Lyhenne tulee sanoista *Discrete Event Simulation*, suomeksi diskreetti tapahtumasimulaatio [21]. Diskreetti tapahtumasimulaatio kohtelee simulaatiota ajan myötä kehittyvänä järjestelmänä. Simulaatio ei velvoita jatkuvaa saapuvaa tapahtumaketjua, vaan simulaatio voi olla toimettomana, kunnes tapahtuma (event) vastaanotetaan. Samaa käytäntöä jalostaen, simulaatio voi ottaa vastaan usean tapahtuman samanaikaisesti, jotka se tyypillisesti suorittaa niiden saapumisjärjestyksessä. [7, s. 10–12.]

Edellä mainitut johtavat juurensa matemaattisesta käsitteestä *dynamic system* (dynaaminen systeemi), joka voi käyttää joko *discrete time* (diskreetti aika) runkoa tai *continuous time* (jatkuva aika, lineaarinen aika) runkoa. Nämä matemaattiset käsitteet ovat tämän opinnäytetyön laajuuden ulkopuolella, joten niitä ei käydä läpi yksityiskohtaisemmin.

Diskreetti tapahtumasimulaatio luo kontrastia verrattuna tyypilliseen aktiviteettipohjaiseen simulaatioon. Aktiviteettipohjainen simulaatio voi joissakin tilanteissa tuhlata paljon prosessorin aikaa, sillä jokaisella simuloidulla ajanhetkellä simulaatiokerneli tarkastelee simulaation tilaa. Jos simulaatiossa määritelty ajan etenemä on erittäin pieni, kuten 0.0001 sekuntia, tulee simulaation eteneminen olemaan hidasta. Tästä voimme muodostaa korrelaation simulaation määritellyn ajan etenemän sekä simulaation suoritusajan välille.

Diskreetti tapahtumasimulaatio hoitaa kyseisen tilanteen järkevämmin siten, että se pitää tallessa kaikki tapahtumat, jotka simulaatiossa tulevat tapahtumaan. Tapahtumia simulaatio vastaanottaa sen jäseniltä, eli moduuleilta. Näistä tapahtumista simulaatio pitää tallessa ajanhetken sekä tapahtuman aiheen. Tämän vuoksi simulaation ei tarvitse tarkistaa jokaisella ajanhetkellä tuleeko jotain tapahtumaan, vaan se voi edetä suoraan seuraavan tapahtuman ajanhetkeen. [7, s. 9.]

4 Aaltomuoto-ohjelmisto

Aaltomuoto-ohjelmisto (myöhemmin aaltomuoto) on opinnäytetyön toimeksiantajan Bittium Wireless:n tuottama ohjelmisto. Aaltomuodon protokollakerrokset on toteutettu C++-ohjelmointikielellä suosien C++ 14 version mukana esiteltyä syntaksia. Vain protokollakerrokset ovat relevantteja tässä työssä, joten niiden ulkopuolisia aaltomuodon komponentteja ei oteta huomioon.

Aaltomuodon ohjelmistoarkkitehtuuri on suunniteltu yhteensopivaksi Bittiumin tuottamalle Tough SDR™ alustalle. SDR lyhentämättömänä on *Software Defined Radio* (ohjelmistolla määritelty radio, ohjelmistoradio). SDR:n tarkoitus on tarjota radioalusta, jonka päälle käyttäjä voi vapaasti ladata aaltomuoto-ohjelmistopaketin. Täten radion toiminta voidaan määritellä täysin ohjelmistollisesti, josta termi SDR on johdettu.

Koska aaltomuodon protokollakerrosten ohjelmistoarkkitehtuuria ei ole erityisesti suunniteltu simuloitavaksi, on oletettavaa, että epäyhteensopivuuksia aaltomuodon ja simulaatioympäristön välillä on olemassa. Näitä epäyhteensopivuuksia käydään läpi yksityiskohtaisemmin myöhemmin dokumentissa.

4.1 Arkkitehtuuri

Aaltomuodon ohjelmistoarkkitehtuurin toteutus ei poikkea yleisestä C++-ohjelmiston arkkitehtuurista kovinkaan paljoa. Arkkitehtuurin muoto on kuitenkin kaukana tavallisesta ohjelmistosta. Tämä johtuu siitä, että ohjelmisto on suunniteltu toimimaan pääsääntöisesti Tough SDR™ alustalla ja Tough SDR™ alusta on suunniteltu täyttämään aaltomuotojen ohjelmistoarkkitehtuurilliset tarpeet.

Aaltomuodon prosessit rakentuvat ohjelmistokomponenteista. Nämä ohjelmistokomponentit ovat tyypillisesti suunniteltu toteuttamaan yhden tehtävän koko ohjelmiston toiminnassa. Jos ohjelmistokomponentti tarvitsee tietoa tai toiminnallisuutta toiselta komponentilta, suurimmassa osassa tapauksista komponenttien välille luodaan funktiorajapinta, jonka kautta komponentit pystyvät viestimään. Toinen yleinen vaihtoehto viestimiseen komponenttien välillä on TCP-socket rajapinta.

4.1.1 Puna/musta-konsepti

Red/Black concept (puna/musta-konsepti, puna/musta-systeemi) on systeemisuunnittelun konsepti, jossa systeemin toiminta erotellaan kahtia: punaiseen ja mustaan osioon. Tyypillisesti systeemin punainen osio käsittelee punaisia signaaleja, jotka sisältävät kryptattua tietoa. Musta osio taas on systeemin julkisempi puoli. Osioiden välissä on usein jonkinlainen mekanismi, joka valvoo osioiden välistä liikennettä, kuten vaikka palomuuri. [22.]

Tough SDR™ alusta tukee puna/musta-konseptia. Täten, aaltomuoto-ohjelmisto voi olla ajossa Tough SDR™ alustalla kyseisessä tilassa [23]. Tough SDR™ alusta on jaoteltu useammalle fyysiselle prosessorille [24]. Prosessoreilla ajossa olevat ohjelmistot kommunikoivat keskenään monin keinoin, joita ei käydä läpi yksityiskohtaisesti tässä dokumentissa.

4.1.2 Singleton-menetelmä

Ohjelmistokomponentit ovat aaltomuodon tapauksessa toteutettu *singleton*-menetelmällä [25]. Singleton-menetelmän mukaan ohjelmistokomponentista luodaan vaan yksi staattinen instanssi, jota sitten hyödynnetään, kun komponenttia halutaan hyödyntää [26]. Täten komponentin käyttö on yksinkertaista, sillä missä vaan samassa nimitilassa sijaitsevassa koodissa voidaan kutsua komponentin metodeja huolimatta ohjelmiston sisällytysrakenteesta.

Singleton-menetelmän toteutus aaltomuodossa on toteutettu niin sanotulla klassisella tavalla. Klassisella tavalla toteutettu singleton-menetelmä on myös yksinkertaisin. Kuvassa 3 nähdään havainnollistava klassinen singletonin toteutus.

```
class Singleton
{
    private:
        /* Here will be the instance stored. */
        static Singleton* instance;

        /* Private constructor to prevent instancing. */
        Singleton();

    public:
        /* Static access method. */
        static Singleton* getInstance();
};

/* Null, because instance will be initialized on demand. */
Singleton* Singleton::instance = 0;

Singleton* Singleton::getInstance()
{
    if (instance == 0)
    {
        instance = new Singleton();
    }

    return instance;
}
```

Kuva 3. Klassinen singleton-menetelmän mukainen toteutus [27].

Klassisessa singleton-menetelmässä luokan konstruktori luodaan yksityisenä, varmistaen ettei mikään muu luokka pääse luomaan oliota luokasta. Täten ainoa luokka, joka voi luoda olion, on luokka itse. Olion luonti tapahtuu kun *getInstance*()-metodia kutsutaan ensimmäisen kerran. Myöhemmillä kerroilla *getInstance*()-metodin kutsuminen palauttaa jo olemassa olevan olion, jonka kautta luokan metodeja voi hyödyntää.

Tämän tavan hyöty on sen yksinkertaisuus toteutuksen kannalta. Menetelmä ei vaadi laajaa perehtymistä ohjelmiston rakenteeseen tai ohjelmointikielen tarjoamiin ominaisuuksiin. Itse toteu-

tus on kovin lyhyt. Tämä tapa ei kuitenkaan tue monisäikeistystä, eli se ei ole thread-safe (säikeytystä varten turvallinen) [28]. Kun toteutustapa ei ole thread-safe, sen käyttäminen monisäikeistettynä voi johtaa määrittelemättömään toimintaan, eli virheisiin.

Aaltomuoto ei kuitenkaan ole monisäikeistetty. Aaltomuodolla voi olla erillisiä tarkkailusäikeitä, joiden tehtävä on vastaanottaa viestejä muilta komponenteilta, mutta nämä säikeet on tarkoituksella eristetty singleton-metodia hyödyntävistä komponenteista. [29.] Täten voidaan taata, että vain yksi säie käyttää singleton-metodilla luotuja olioita saman aikaisesti.

Singleton-metodi ohjelmiston luomiseen on melko yleinen tapa olio-ohjelmointipohjaisissa ohjelmointikielissä. Aaltomuodon lähdekoodi on pääosin C++-ohjelmointikielellä tuotettua, joten ohjelmistoarkkitehtuurin kannalta päätös käyttää singleton-metodia on perusteltua.

4.1.3 Reaaliaikaisuus

Aaltomuoto muodostaa ohjelmistokomponenteillaan lineaarisen polun datalle, jota se on joko lähettämässä tai vastaanottamassa. Polun ytimenä toimii aaltomuodon *slot processing cycle* (tilaprosessointisykli), jonka avulla aaltomuoto hallitsee lähetystä ja vastaanottoa. [30.] Tilaprosessointisyklin toiminta on erittäin kriittinen ohjelmiston oletetun toiminnallisuuden kannalta. Tilaprosessointisyklin toiminta vaatii tarkkoja ajastimia sekä reaaliaikaista kelloa alustalta. Ohjelmiston toiminta nojaa vahvasti siihen olettamukseen, että ajastimet ja kellot, joita ohjelmisto käyttää ovat riittävän tarkkoja, jotta niiden toiminta/toimimattomuus eivät vaikuta kriittiseen toiminnallisuuteen. [31.]

Aaltomuodon ohjelmistollinen toiminta vaatii alustalta tarkkaa aikaa sekä reaaliaikaista suorituskykeneväisyyttä. Nämä tulee onnistua myös OMNeT++-simulaatioympäristöltä jos aaltomuoto halutaan ympäristöön ajoon. Epätarkat ajoitukset tai muut ajoitukseen liittyvät hankaluudet OMNeT++:n puolelta voivat muodostua suureksi esteeksi aaltomuodon simuloinnille.

4.2 Simulaatiokykeneväisyys

Kuten aiemmassa kohdassa jo todettiin, aaltomuodon ohjelmistoarkkitehtuuria ei ole suunniteltu simulointia varten. OMNeT++-simulaatioympäristö on alustana kovin erilainen aaltomuodolle

verrattuna Tough SDR™ alustaan, mainitsemattakaan sitä kuinka aaltomuoto on erityisesti suunniteltu Tough SDR™ alustalle. Erilaiset haasteet ja epäyhteensopivuudet olivat oletettuja prosessin aikana.

Ennen simulaation toteutuksen aloitusta tehtiin arvio siitä, tulisiko aaltomuoto toimimaan simulaatiossa. Arvio perustui tutkimustyöhön koskien OMNeT++:n ohjelmistoarkkitehtuuria sekä aaltomuodon ohjelmistoarkkitehtuuria. Tutkimustyön tehneenä todettiin, että vaikka aaltomuoto voisi toimia OMNeT++-simulaatioympäristössä, esteeksi muodostuisi todennäköisesti OMNeT++:n ohjelmistoarkkitehtuurin asettamat vaatimukset simuloitavalle ohjelmistolle. Alustojen rakenteellisten erojen takia simuloinnin aikana kohdattaisiin varmasti epäyhteensopivuuksia. Kysymykseksi jäikin ennemminkin se, pystyttäisiinkö löydetyt ongelmat kiertämään tai selvittämään ilman suuria muutoksia ohjelmistoon.

5 Yhteensopivuus

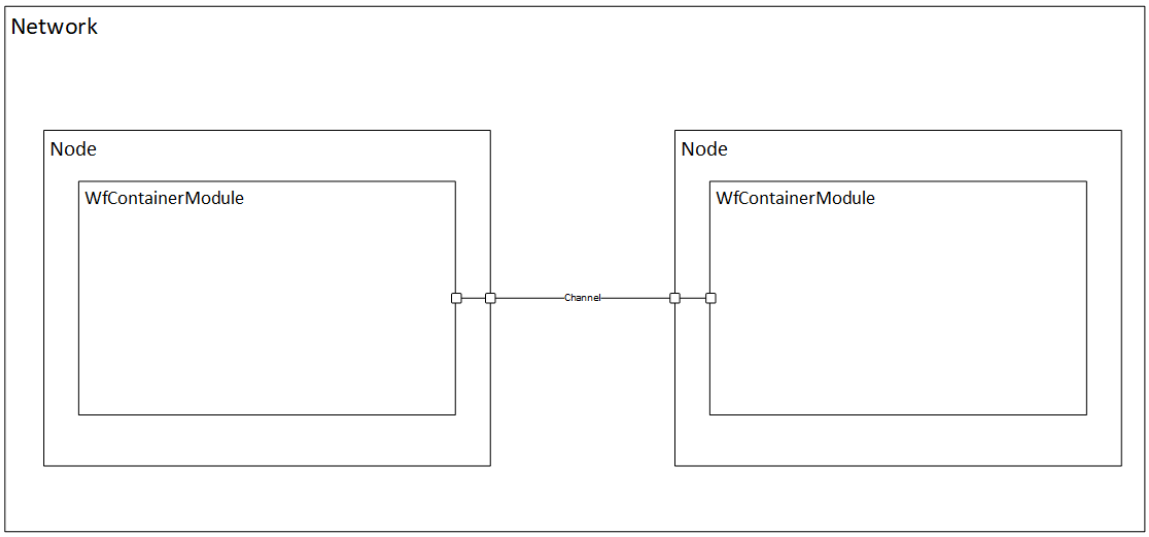
Ohjelmistojen arkkitehtuurien yhteensopivuus ja siihen vaikuttavat tekijät olivat työssä selvitettävänä tekijänä. Tässä luvussa esitellään ne tekijät, jotka löydettiin vaikuttavan ohjelmiston yhteensopivuuteen, kun ohjelmistoa koetetaan käyttää laiteympäristössä, eli Tough SDR™ alustalla sekä OMNeT++-simulaatioympäristössä.

Yhteensopivuuteen vaikuttavat tekijät olivat pääsääntöisesti ohjelmistoarkkitehtuurillisia tekijöitä. Jos nämä tekijät olisivat tiedossa jo ohjelmiston arkkitehtuurin suunnitteluvaiheessa, ne voitaisiin välttää tai vähintäänkin niiden vaikutuksia voitaisiin hillitä.

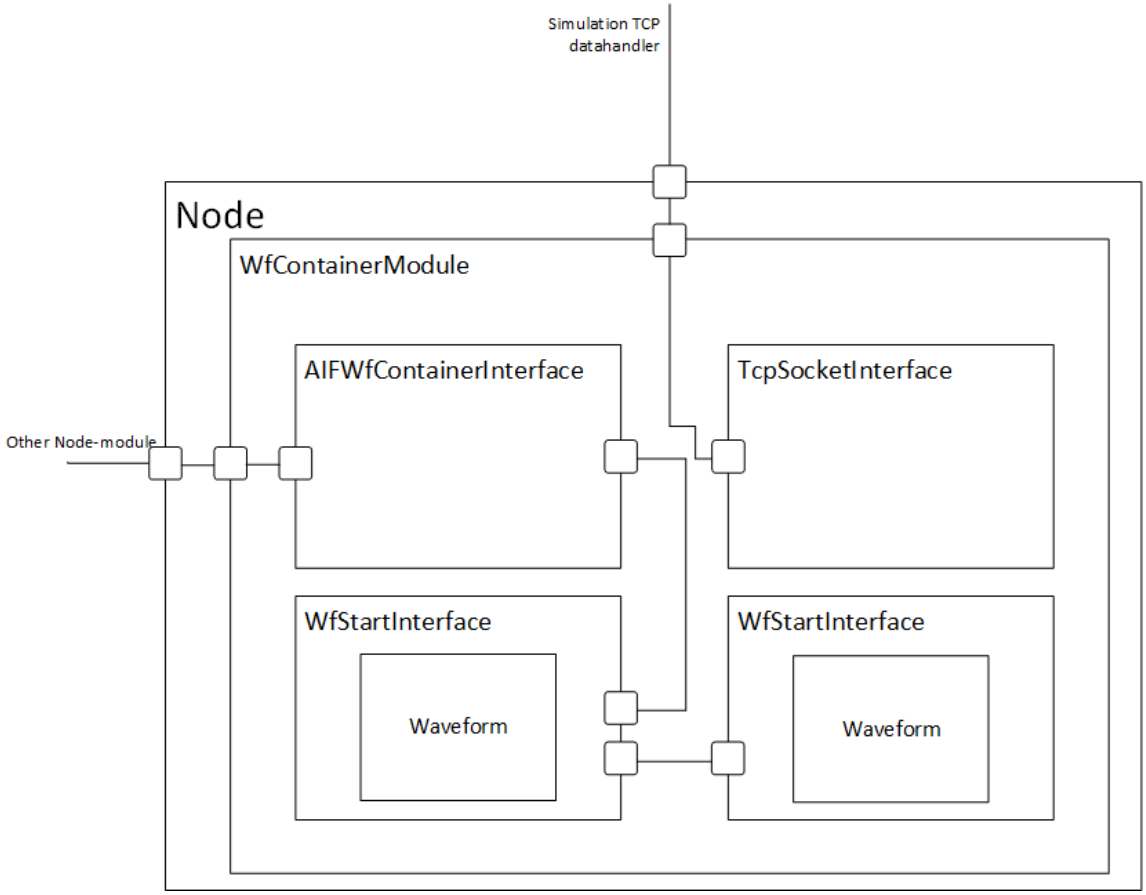
5.1 Toteutettu simulaatiomalli

Yhteensopivuutta tavoiteltiin sovittamalla aaltomuoto suoraan OMNeT++-simulaatioympäristöön. Sovittaminen, josta käytetään myös termejä tuominen tai porttaaminen, tarkoittaa ohjelmiston tuomista ympäristöön, johon sitä ei ole suunniteltu sekä ympäristön aiheuttaminen mahdollisten epäyhteensopivuuksien ja vikojen korjaamista.

Simulaatiomallin moduulihierarkian ylimmälle tasolle luotiin yhdistelmämoduuli *Node*. Node on termi, jota tyypillisesti käytetään kommunikaatioverkoista puhuessa kuvaamaan yksittäistä laitetta, niin sanotusti solmua verkossa. Luodun simulaatiomallin korkeatasoinen rakenne näkyy kuvasta 4, jossa näkyy simulaation keskenään viestivät Node-moduulit. Nodet viestivät käyttäen *channelia* (kanavaa), jolle voidaan määritellä erilaisia ominaisuuksia. Node-moduulin tarkka rakenne sekä sisältämät moduulit ovat näkyvissä kuvassa 5. Node sisältää yhden yhdistelmämoduulin *WfContainerInterface* (Waveform Container Interface, aaltomuodon konttirajapinta). Node-moduulin portit (gates) on yhdistetty sen sisältämään WfContainerInterfacen portteihin. Tämä mahdollistaa sen, että Node-moduulin ilmarajapintaa mallintava portti voi vastaanottaa dataa toisilta Node-moduuleilta ja ohjata sen alaspäin moduulihierarkiassa. Ilmarajapintaa mallintava portti on määritelty niin, että se sisältää @directin määritelmän. Tämä mahdollistaa sen, että portti vastaanottaa viestejä, jotka ovat lähetetty *sendDirect*()-metodia käyttämällä, joka on usein suosittu tapa radioiden ilmarajapintoja simuloidessa. [32.]



Kuva 4. Simulaatiomallin korkean tason hierarkia kuvattuna.



Kuva 5. Node-moduulin rakenne ja sen sisältämät yksinkertaiset moduulit.

WfContainerInterface-moduuli omistaa useamman yksinkertaisen moduulin. Moduulin määritelmä näkyy kuvasta 6. Moduulin alimoduuleihin kuuluu *AIFWfContainerInterface* (Air Interface Waveform Container Interface, ilmarajapinnan aaltomuodon konttirajapinta), kaksi instanssia

moduulia *WfStartInterface* (Waveform Start Interface, aaltomuodon käynnistysrajapinta) sekä *TCPSocketInterface* (TCP-socket rajapinta).

```
module WfContainerModule
{
    gates:
        inout g[];
    submodules:
        aifInterface: AIFWfContainerInterface;
        wfInterfaceL: WfStartInterface;
        wfInterfaceH: WfStartInterface;
        tcpIf: TcpSocketInterface;
    connections:
        for i=0..sizeof(g)-1 {
            g++ <--> aifInterface.g++;
        }
}
```

Kuva 6. WfContainerInterface-moduulin määritelmä NED-koodikielellä.

AIFWfContainerInterfacen portit ovat liitetty WfContainerInterfacen portteihin molempiin suuntiin. Liitokset näiden moduulien välillä sekä liitokset WfContainerInterfacen ja Node-moduulin välillä muodostavat simulaation AIF-rajapinnan. Kun Node-moduuli vastaanottaa viestin, viesti ohjataan ensin WfContainerInterfacelle, joka ohjaa sen edelleen AIFWfContainerInterfacelle. Koska AIFWfContainerInterface on yksinkertainen moduuli, viestin käsittely siirtyy moduulin C++-koodille. C++-koodi käsittelee viestin ja poistaa siitä mahdolliset simulaation vaatimat ylimääräiset tavut. Kun vain aaltomuodon lähettämä data on jäljellä viestistä, se ohjataan aaltomuodon vastaanottavaan ilmarajapintaan. Toimimalla näin, aaltomuoto ei ole tietoinen siitä, että ohjelmistoa ajetaan simulaatioympäristössä, vaan käsittelee viestin identtisesti verrattuna Tough SDR™ alustalla ajoon. Aaltomuodon sisäisten ja ulkoisten rajapintayhteyksien abstraktointi onkin yksi olennaisista tekijöistä yhteensopivuuden tavoittelussa aaltomuodon ja vaihtelevien alustojen välillä. Muita tekijöitä käydään läpi myöhemmin dokumentissa.

Simulaatioskenaario muodostuu kahdesta Node-moduulista, jotka viestivät keskenään käyttäen aaltomuotoa. Simulaation alkuperäinen tavoite oli simuloida aaltomuodon CSMA-algoritmeja, joten aaltomuodon asetuksia tuli saada muokattua, jotta erilaisia CSMA-algoritmeja voitaisiin valita. Tämä onnistui OMNeT++:n tarjoamilla INI-tiedostoilla, joista asetetut parametrit ladataan moduulien kautta aaltomuodolle simulaation alkaessa. Aaltomuodon komponentit käynnistetään yksilöllisillä instansseilla WfStartInterface-moduulissa. Moduulin *initialize()*-metodissa luodaan uusi säie, joka kutsuu aaltomuotokomponentin muokattua *main()*-metodia.

5.2 Ohjelmistojen rajoitteet

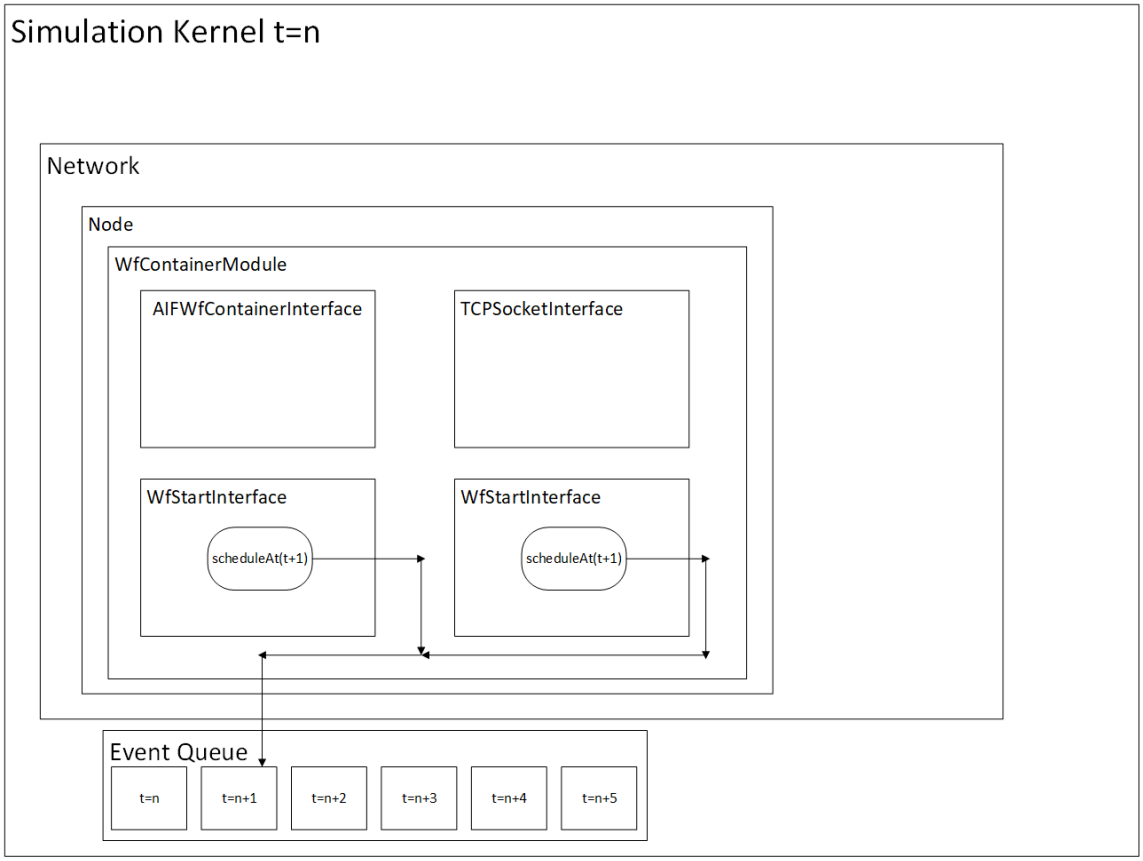
Aaltomuodon sekä OMNeT++:n ohjelmistoarkkitehtuurien asettamat rajoitteet ilmenivät simulaatiomallin toteutuksen aikana niin, että simulointi todettiin mahdottomaksi tai vähintään erittäin epäkäytännölliseksi. Ongelma tiivistyi OMNeT++-simulaatioympäristön käyttöön alustana aaltomuodolle, jonka ohjelmistoarkkitehtuuri on taas sen sijaan suunniteltu toimivaan sulavasti Tough SDR™ alustalla. Eroavaisuudet Tough SDR™ alustalla sekä OMNeT++-simulaatioympäristöllä alustana johtivat laajamittaisiin epäyhteensopivuuksiin.

5.2.1 OMNeT++

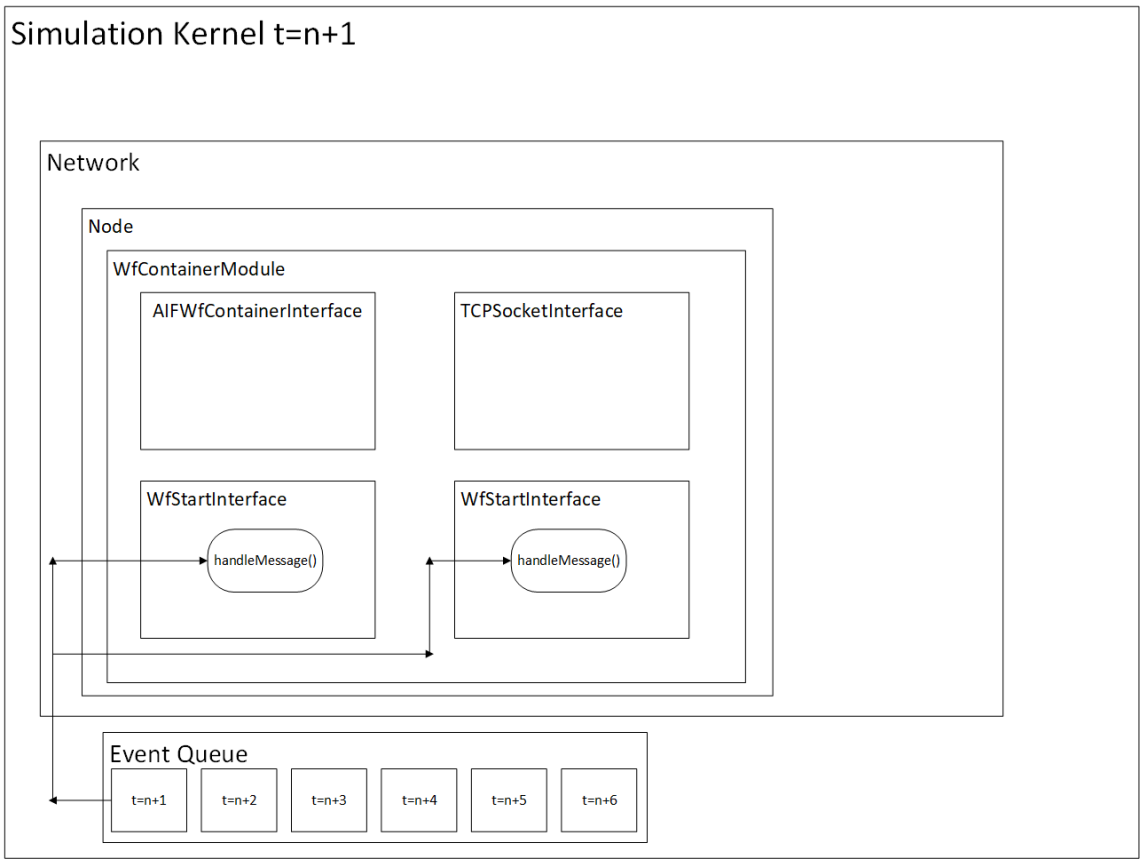
OMNeT++:n tarjoama joustavuus simulaatiomallia luodessa voi yllättää, kunnes muistaa, että ohjelmiston nimikin on Objective Modular Network Testbed (objektiivisesti modulaarinen verkkotestausympäristö). OMNeT++ tarjosi monenlaisia ratkaisuja yhteensopivuusongelmiin. Ratkaisut tyypillisesti muodostuivat simulaatiokirjaston ominaisuuksien hyödyntämisestä omaan käyttötarkoitukseen hyödyntämällä C++-periytymisperiaatteita. OMNeT++ tukee laajasti periyttämistä sen sisäisistäkin komponenteista tiettyjen rajoitusten kera [33]. OMNeT++ voi tukea laajaa periyttämistä sen sisäisistä komponenteista, sillä sen ohjelmistoarkkitehtuuri on suunniteltu sitä varten. Täten OMNeT++ on erityisen modulaarinen ohjelmisto. OMNeT++:n kykeneväisyydet erilaisten ohjelmistojen simulointiin ovat suoraan riippuvaisia siitä, kuinka paljon käyttäjä on valmis luomaan muokattua toiminnallisuutta OMNeT++:n tarjoamista systeemikomponenteista.

Aaltomuodon ajastintoteutukset eivät toimineet OMNeT++-simulaatioympäristössä ilman muokkausta, sillä ne hyödynsivät Linux-käyttöjärjestelmän tarjoamia funktioita C++ ja C-ohjelmointikielien rajapintojen kautta. Tämä ongelma ratkaistiin melko yksinkertaisesti OMNeT++:n keinolla muodostaa ajastimia käyttäen moduulien itselleen lähettämiä viestejä (self-messaging) [34]. Moduulien itselleen lähettämät viestit ovat melko yksinkertainen konsepti. Mikä tahansa yksinkertainen moduuli voi C++-koodissaan kutsua OMNeT++:n ohjelmointiraamin tarjoamaa metodia *scheduleAt(simtime_t time, cPacket packet)*. Tämä metodi ottaa vastaan ajan ja paketin, joka tulee lähettää. Kun simulaation nykyinen aika vastaa metodille annettua aikaa, metodille annettu paketti lähetetään sille moduulille, jonka C++-koodi on metodia kutsunut. Moduuli lähettää siis itselleen viestin. Kun viesti saapuu moduulille, sen käsittely siirretään moduulin C++-koodille. Tyy-

pillistä on, että tässä vaiheessa toteutusta tarkistetaan mikä ajastin lähetti kyseisen paketin. Tarkistus on erityisen tärkeää siinä tilanteessa, jos kyseessä on ohjelmisto, jolla on useampi ajastin. Itselleen lähetettyjen viestien toimintaa simulaatiotilassa havainnollistetaan kuvilla 7 ja 8.



Kuva 7. Self-messagen ajastaminen ajanhetkellä t=n käyttäen *scheduleAt()* metodia.



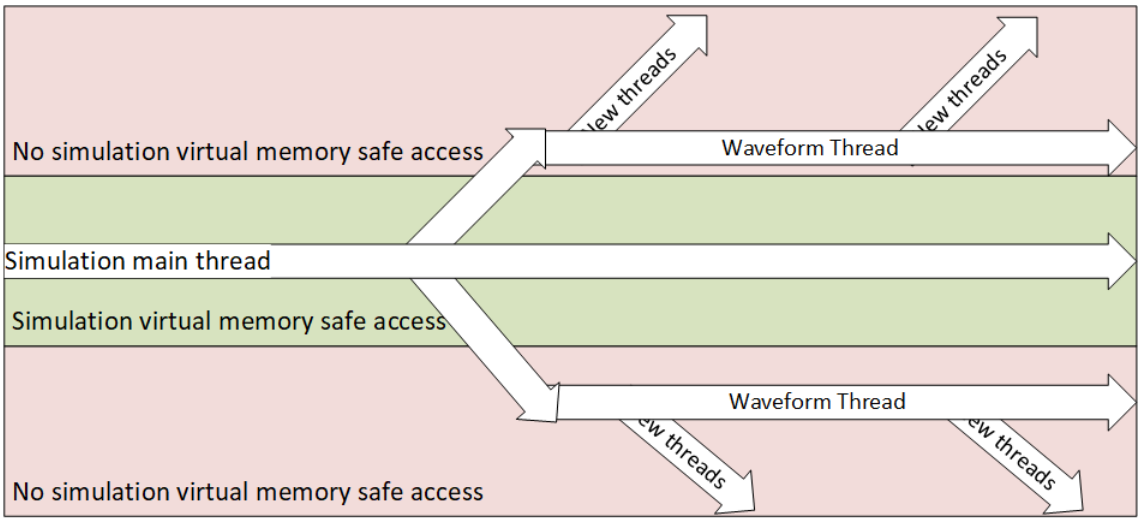
Kuva 8. Self-messagen saapuminen ajanhetkellä t=n+1 moduulin *handleMessage()* metodiin simulaatiokerneliltä.

OMNeT++:n tarjoamien metodien ohjelmointiperiaatteet asettavat myös omia rajoituksiaan simuloitavalle ohjelmistolle. Yksi näistä rajoituksista liittyy säikeiden käyttöön. OMNeT++ API:n tarjoamat metodit eivät ole *thread safe* (monisäikeistys-turvallisia), joka käytännössä estää niiden käytön monisäikeistetyssä ympäristössä [35]. Jos säikeitä tästä huolimatta käytettäisi näiden metodien kutsumiseen, metodien toiminta tulee olemaan määrittelemätöntä, johtuen mahdollisesta ongelmatilanteesta, jossa usea säie muokkaa samaa muistitilaa. On kuitenkin kannattavaa huomata, että monisäikeistys-turvallisuuden tietolähde ei ole virallinen OMNeT++:n julkaisema tieto, joten lähteen luotettavuus on vähintäänkin kyseenalainen. Väitettä vääräksi todistavia kohtia ei kuitenkaan löydetty OMNeT++:n koodista.

Monisäikeistykseen liittyvää ongelmaa voidaan koettaa torjua muutaman keinon avulla. Yksi näistä on yksinkertaisesti olla kutsumatta säikeissä metodeja, jotka eivät ole monisäikeistys-turvallisia. Täten säikeitä voidaan luoda useampia, kunhan käyttäjä voi varmistaa, etteivät säikeet pääse käsiksi kyseisiin metodeihin. Tämän vuoksi aaltomuodon viestejä vastaanottavat säikeet

olivat sallittuja simulaatiossa, koska ne eivät käsittele OMNeT++ API-metodeja tai muokkaa simulaation tilaa.

Kyseisellä lähestymistavalla menetellessä, aaltomuodon komponenttien käynnistäminen tulee delegoida erillisille säikeille. Jos delegointia ei tehtäisi, OMNeT++:n luoman alkuperäisen säikeen hallinta ei palaisi koskaan simulaatiolle, vaan jäisi aaltomuodon hallintaan. Näin käy koska aaltomuodolla on *main loop* (ohjelmiston pääsilmukka) joka päivittää aaltomuodon tilaa jatkuvasti. Kun säie saavuttaa ohjelmiston pääsilmukan, se jää niin sanotusti pyörimään silmukkaan siihen asti, että aaltomuoto saa signaalin keskeyttää ajon. WfStartInterface-moduuleissa aaltomuodon komponentit käynnistetään omissa säikeissään. Kuvassa 9 nähdään, että koska komponentit ovat ajossa muissa säikeissä kuin simulaation pääsäikeessä, ne eivät saa käyttää tai muokata OMNeT++ API-metodeja tai simulaation muistitilaa. Tuloksena on lähinnä päänvaivaa ohjelmoijalle.



Kuva 9. OMNeT++-simulaatioprosessin virtuaalisen muistitilan käytön turvallisuus säikeille.

5.2.2 Aaltomuoto

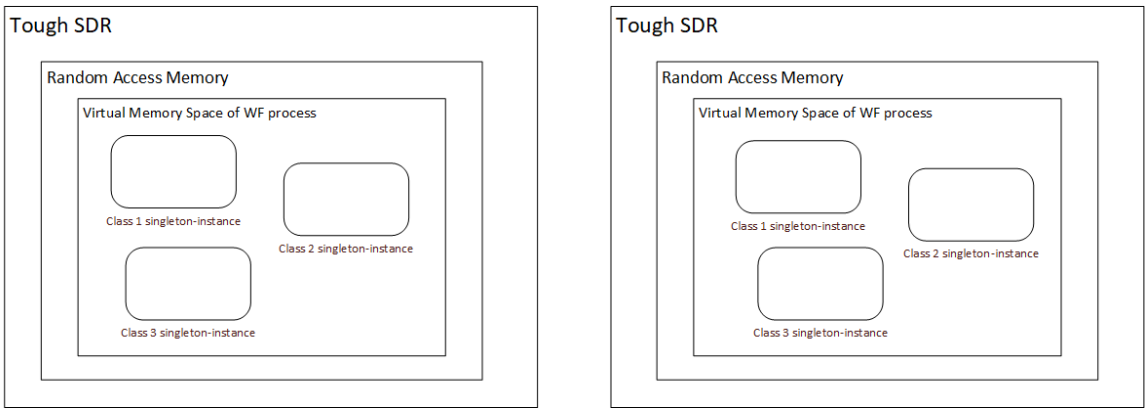
Aaltomuodon ohjelmistoarkkitehtuurin rakenteen ja ominaisuuksien priorisoinnin takia sen yhteensopivuus OMNeT++-simulaatioympäristöön oli vähemmän kuin optimaalinen. Ongelmallisuus oli kuitenkin oletettavissa, sillä kuten dokumentissa on jo useaan kertaan mainittukin, aaltomuodon ohjelmistoarkkitehtuurinsuunnittelussa ei ole otettu huomioon ohjelmiston simulaatiokykeneväisyyttä, tai varsinkaan yhteensopivuutta OMNeT++-simulaatioympäristössä ajoa var-

ten. Aaltomuodon ohjelmistoarkkitehtuuri olisi vaatinut refaktorointia toimiakseen simulointitarkoituksiin. Opinnäytetyön tavoite tiivistyikin tässä vaiheessa siihen, minkälaiset tekijät suunnitteluvaiheessa voisivat poistaa tämän refaktoroinnin tarpeen ohjelmistolta.

Suurimmaksi ongelmaksi muodostui aaltomuodon singleton-menetelmän laaja käyttö ohjelmistokomponenteissa. Singleton-menetelmä pohjautuu staattisten ohjelmointirakenteiden käyttöön. Ohjelmointirakenteet ovat tyypillisesti metodi tai metodeja, jotka varmistavat, että vain yksittäinen olio ohjelmistokomponentista on olemassa samanaikaisesti.

Kuvassa 9 nähdään havainnollistus siitä, miten aaltomuoto on jakautunut ollessaan ajossa Tough SDR™ alustalla. Aaltomuodon prosessit ovat ajossa erillisillä fyysisillä prosessoreilla laitteella. Koska aaltomuodon eri ohjelmistot ovat ajossa eriytetyissä prosesseissa, ne käyttävät prosessikohtaista *virtual address space*a (virtuaalinen muistitila). [36.] Prosessin muistitila on käyttöjärjestelmän rajaama alue käyttömuistista, jossa ohjelma säilyttää tarvitsemansa muuttujat ja metodit. Singleton-menetelmää hyödyntävät aaltomuodon ohjelmistokomponentit määrittelevät yhden staattisen metodin: *getInstance()*. Tämän metodin tarvitsema tila virtuaalisessa muistitilassa on aina allokoitu. Metodia voi siis hyödyntää vaikkei yhtäkään oliota sen luokasta ole luotu [37].

Koska aaltomuodon prosessit omaavat yksilölliset virtuaaliset muistitilat, ne voivat vapaasti hyödyntää singleton-menetelmän mukaisia komponentteja. Muistitilat prosessien välillä eivät limity käyttömuistissa, joten prosessit voivat hyödyntää staattisuutta. Sen lisäksi, että erilliset prosessit omaavat yksilölliset virtuaaliset muistitilat, prosessit ovat ajossa eri fyysisillä prosessoreilla. Kuvasta 10 nähdään myös, että kahden aaltomuodon välinen viestintä on mahdollista, sillä aaltomuoto on ajossa erillisillä fyysisillä laitteilla.

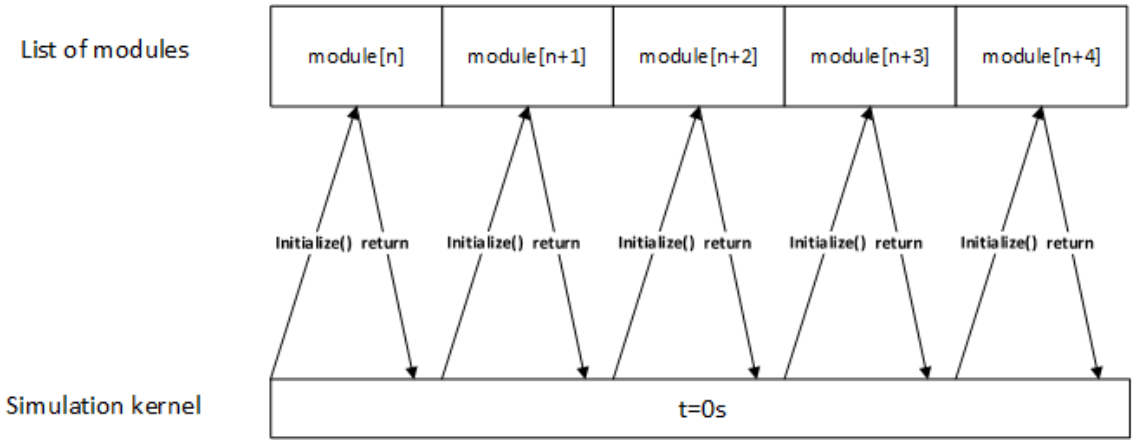


Kuva 10. Kahden Tough SDR™ alustan aaltomuotojen prosessien singleton-instanssit, jossa singleton-menetelmän käyttö on ongelmatonta.

Simulaatioympäristössä mikään osa äsken kuvailluista periaatteista ei päde. Kuten OMNeT++ omassa artikkelissaan kertovat C++-ohjelmistojen tuomisesta OMNeT++-simulaatioympäristöön: ”*If the library is well-designed, one can just create as many instances of the network stack (or the component is question) as needed. Except of course, if the library makes use of the dreaded Singleton pattern, or otherwise uses global variables.”* [34]. Toisin sanoen, jos ohjelmiston arkkitehtuuri pohjautuu singleton-menetelmän hyödyntämiseen, ohjelmiston tuominen OMNeT++-simulaatioympäristöön tulee olemaan haastavampaa.

OMNeT++:n julkaiseman artikkelin tapaan kuvailla singleton-menetelmää on syynsä. OMNeT++:n ohjelmistoarkkitehtuuri on rakennettu sille olettamukselle, että simulaatio on ajossa yksittäisessä prosessissa. Kyseinen periaate on juurtunut erittäin syvälle ohjelmiston toimintaan ja suuri osa jatkokehityksestä alkuperäisen version jälkeen on tapahtunut tämän olettamuksen alaisena. OMNeT++ tarjoaa runsaasti keinoja muokata simulaatiokernelin toimintaa. Tästä huolimatta, simulaatioympäristöstä yksittäiseen prosessiin riippuvuuden poistaminen on todennäköisesti mahdotonta tai vähintäänkin erittäin työlästä ja epäkäytännöllistä. Vaikka yksittäiseen prosessiin riippuvuus saataisiin poistettua simulaatioympäristöstä, kun simulaatioympäristöä haluttaisiin päivittää tulevaisuudessa, tulisi samat muutokset toteuttaa uudestaan.

Ongelmatilanne muodostuu yksinkertaisesti ohjelmistojen arkkitehtuurien asettamien vaatimusten vastakkainasettelusta. OMNeT++-simulaatio tulee aina olemaan ajossa vain yhdessä prosessissa samanaikaisesti. Aaltomuodon ohjelmisto on suunniteltu toimimaan kahdessa prosessissa erillisillä laitteilla. Täten, kun yllä kuvailtu simulaatioskenaario käynnistetään, vain aaltomuodoista ensimmäinen käynnistyy oletetusti. Ongelma johtaa juurensa siihen, miten ohjelmistoprosessit käyttäytyvät. OMNeT++ luo simulaation sisältämät moduulit simulaation käynnistyksen aikana alla olevan kuvan 11 mukaisesti. Kuva havainnollistaa, kuinka OMNeT++:n tiedossa olevien moduulien listalta indeksillä 0 kutsutaan moduulin *initialize()* metodia. Ei ole tiedossa, miten OMNeT++:n simulaatiokerneli luo edellä mainitun listan simulaatiomoduuleista. [38.]



Kuva 11. OMNeT++:n simulaatiokernelin käynnistyssekvenssi.

Kun moduulien listalta indeksillä 0 kutsutaan moduulin *initialize()* metodia, ensimmäinen Node-moduuli simulaatiomallissa käynnistyy. Tämä puolestaan käynnistää aaltomuodon, joka luo singleton-menetelmää hyödyntävät ohjelmistokomponenttiolionsa. On tärkeää huomioida, että tässä vaiheessa käynnistymissekvenssiä, yksi moduuli on luonut staattiset oliot aaltomuodon komponenteista. Kun moduuli indeksillä 0 on saanut *initialize()* metodinsa suoritettua, moduulin indeksillä 1 *initialize()* metodia kutsutaan simulaatiokernelistä. Kun simulaatiomallin toinen Node-moduuli aloittaa *initialize()* metodinsa ja täten aaltomuotonsa käynnistyksen, aaltomuoto käsittää olevansa jo käynnissä. Node-moduulin indeksillä 1 aaltomuoto näkee staattiset olionsa jo olemassa, jotka todellisuudessa ovat Node-moduulin indeksillä 0 luomia. Tilanne johtaa siihen, että Node-moduulin indeksillä 1 aaltomuoto ilmoittaa olevansa käynnistynyt, vaikka todellisuudessa aaltomuodot molemmissa Node-moduuleissa ovat käytännössä identtiset. Kaksi Node-moduulia eivät voi viestiä keskenään aaltomuodon avulla, sillä aaltomuodot ovat yksi ja sama.

Aaltomuodon ohjelmistoarkkitehtuurin aiheuttamista ongelmista suurin on singleton-menetelmän hyödyntäminen ohjelmistossa. OMNeT++:n artikkeli ehdottaa tämänkaltaisten ohjelmistojen simulaatioympäristöön tuomista varten seuraavanlaista: ”*However, singletons are usually easy to identify in the code, and the source can be modified to allow several instances to coexist in memory*.” [34.] Singleton-menetelmä tulisi OMNeT++-simulaatioympäristöön ohjelmistoa tuodessa korvata muulla menetelmällä. Ongelmaksi kyseissä lähestymistavassa muodostuukin ohjelmiston koko ja kompleksisuus, joista molempien määrä vaikuttaa ohjelmiston muokkaamiseen vaadittuun työmäärää eksponentiaalisesti. Täten on suotavaa vetää johtopäätös, että singleton-menetelmää ei tulisi hyödyntää ohjelmiston arkkitehtuurissa, jonka toimintaa on tarkoitus simuloida OMNeT++-simulaatioympäristössä.

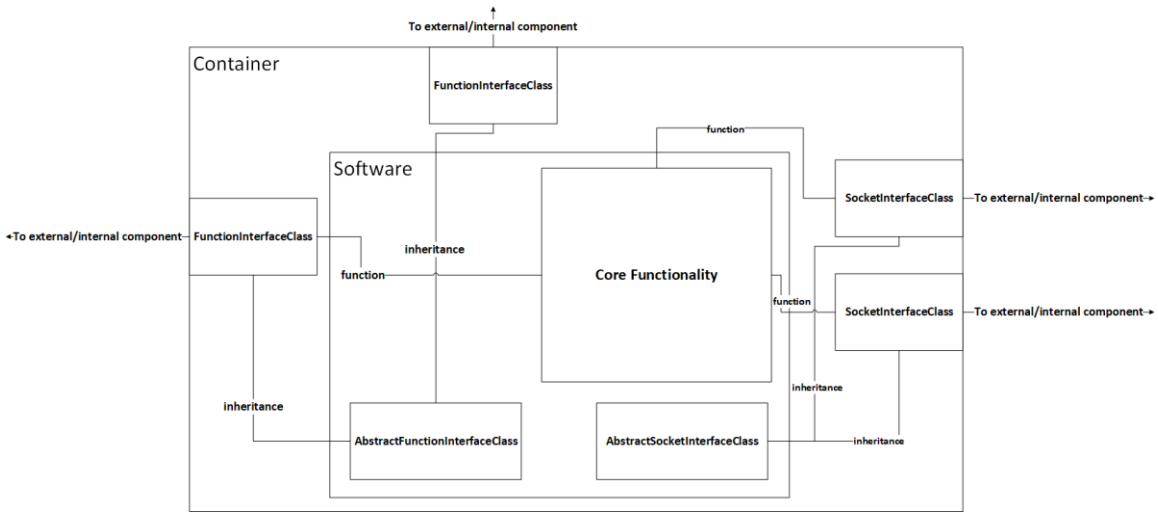
Toinen ongelma aaltomuodon ohjelmistoarkkitehtuurissa on laaja riippuvuus socket-interfaceihin (verkkopistokerajapintoihin). Verkkopistokkeet eri ohjelmistokomponenttien välillä mahdollistavat yksinkertaisen viestinnän. Ongelma muodostuu siitä, että käyttöjärjestelmä sallii vain yhden uniikin verkkopistokkeen olemassaolon samanaikaisesti. Viisi muuttujaa tekevät verkkopistokkeesta uniikin: paikallinen IP, paikallinen porttinumero, etä-IP, etäporttinumero sekä protokolla (TCP tai UDP). Jos jokin näistä muuttujista on arvoltaan erilainen kuin muissa käyttöjärjestelmän aktiivisissa verkkopistokkeissa, se on uniikki. Muussa tapauksessa verkkopistoke ei voi toteuttaa tehtäväänsä. [39.] Kun aaltomuodon ensimmäinen käynnistyvä instanssi luo verkkopistokkeensa aaltomuodon käyttämillä porttinumeroilla, myöhemmin käynnistyvät aaltomuodon instanssit eivät täten enää pysty luomaan tarvitsemiansa verkkopistokkeita. Useat aaltomuodon instanssit pystyvät käyttämään samoja porttinumeroita ollessaan ajossa Tough SDR™ alustalla, sillä jokainen laite on käytännössä erillinen käyttöjärjestelmäympäristö. Tämä ei kuitenkaan päde simulaation tapauksessa, sillä useampaa aaltomuotoinstanssia yritetään luoda yksittäisen käyttöjärjestelmän sisäisesti.

5.3 Konttiarkkitehtuuri

Jotta edellä kuvailtuja ohjelmistojen rajoitteita voitaisiin ennakoivasti torjua ohjelmiston suunnitteluvaiheessa, on tarpeellista havainnollistaa miten ohjelmisto voisi sopia sekä laite- että simulaatioympäristöön minimaalisilla muutoksilla. Tässä kappaleessa käydään läpi kaksi erilaista ohjelmistoarkkitehtuuria, joiden soveltaminen kehityksessä voi mahdollistaa ohjelmiston simuloinnin. Arkkitehtuurien yhdistävä tekijä on, että niistä molemmat luovat *indirectionia* (epäsuoruutta) eri ohjelmistokomponentteihin suuntautuviin yhteyksiin. Epäsuoruus ohjelmistoarkkitehtuurisuunnittelussa perustuu ajatukseen, että ohjelmistokomponentin ei tarvitse olla tietoinen siitä, miten data kulkeutuu vastaanottavalle komponentille. Komponentin tarvitsee vain olla tietoinen siitä mihin data on lähetettävä. Komponentti antaa datan rajapinnalle, joka päättelee ajoympäristön mukaan datan lähetykseen sopivan keinon.

Ohjelmistokomponentilta ulospäin suuntautuvien rajapintojen abstraktointi muodostaa eräänlaisen kontin ohjelmiston ympärille, kuten kuvasta 12 voi todeta. Ohjelmisto voi kommunikoida ulkopuolisten komponenttien kanssa mutta kaikki yhteydet ohjelmistolta ulospäin kulkevat kontin tarjoamien väylien avulla. Kontin rajapintojen ei tarvitse olla rajoittunut vain ohjelmistolta muille

ohjelmistoille, vaan se voi tarjota myös ohjelmiston sisäisten komponenttien välisiä yhteysrajapintoja. Englannin kielellä kyseinen ohjelmistoarkkitehtuuri on nimeltään *container architecture* (konttiarkkitehtuuri)



Kuva 12. Abstrakti arkkitehtuurikuva konttiarkkitehtuurista.

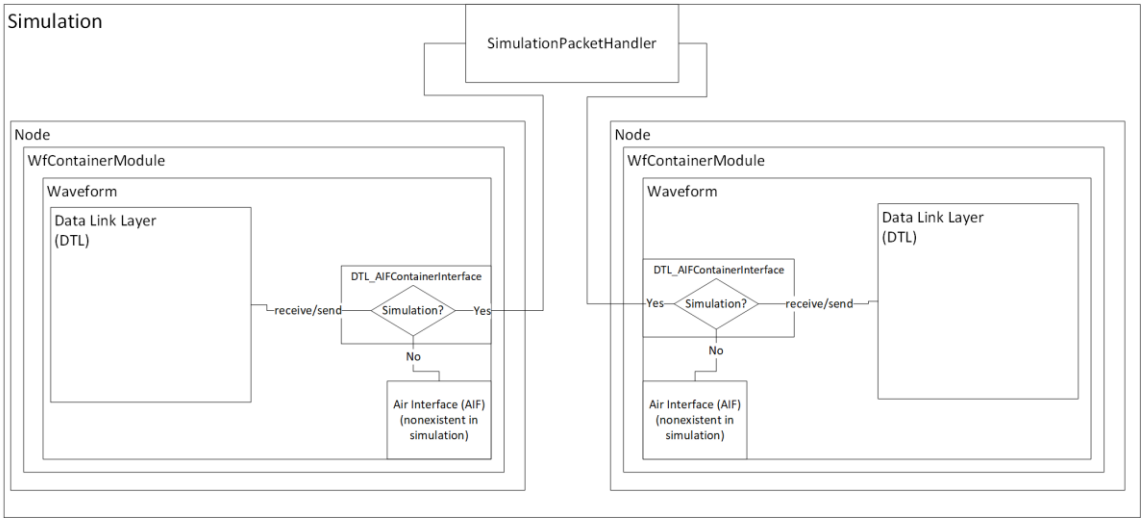
5.3.1 Aaltomuotokonttiarkkitehtuuri

Waveform Container Architecture (aaltomuotokonttiarkkitehtuuri) on ensimmäinen ohjelmistoarkkitehtuurimalleista, joiden on tarkoitus estää ongelmia ohjelmiston yhteensopivuudessa OMNeT++-simulaatioympäristöön sekä laiteympäristöön. Arkkitehtuurimalli perustuu ajatukseen, että aaltomuodon ja simulaatiomallin välissä on container (kontti), jonka tarkoitus on liittää kaksi ohjelmistoa toisiinsa. Samankaltaista ideaa ajetaan myös OMNeT++:n artikkelissa, jossa käytetään termiä *glue code* (liimakoodi). Liimakoodi-termillä tarkoitetaan koodia, jonka tarkoitus on liittää kahden erillisen ohjelmiston toiminta toisiinsa. Liimakoodi ei kuitenkaan ole käytännössä sama asia kuin kontti. Kontin tarkoitus on yhdistää kahden ohjelmiston toiminta mutta myös tarjota abstraktiota aaltomuodolle. Kontti ei ole siis ainoastaan simulaatioympäristössä hyödynnettävää kovakoodattua liimakoodia, vaan monikäyttöinen ohjelmistorajapintakerros, joka ympäröi aaltomuodon ydintoteutusta.

Kontin toteutus tiivistyy käytännössä rajapintojen tarjoamiseen. Koska aaltomuoto muodostuu C++-koodista, on sopivaa olettaa, että myös kontin toteutus muodostuu C++-koodista. Kontin tulee sisältää rajapintaluokka jokaista aaltomuodon käyttämää yhteyttä varten. Aaltomuodon ydintoteutuksessa tulee hyödyntää näiden rajapintaluokkien tarjoamia metodeja. Rajapintaluokkien

tulee sisältää metodeja, joiden avulla aaltomuodon ydintoteutus voi lähettää ja vastaanottaa dataa muilta sen tarvitsemilta komponenteilta riippumatta siitä, missä ympäristössä aaltomuoto on ajossa.

Kuvassa 13 nähdään, miltä teoreettinen datan polku ohjelmiston läpi näyttäisi. Kuvan tapauksessa aaltomuodon DTL-kerros (Data Link Layer) lähettää dataa toiselle radiolle, jossa DTL-kerros vastaanottaa paketin. DTL-kerroksen yhteys AIF-kerrokselle (Air Interface) on abstraktoitu kontin tarjoaman rajapinnan ansiosta, joten DTL-kerros ei ole tietoinen toimintaympäristöstään. Kun DTL-kerros kutsuu dataa lähettävää metodia sen rajapintaoliossa, data kulkeutuu rajapinnan koodille. Rajapintakoodi sen sijaan on tietoinen siitä, että ohjelmisto on ajossa simulaatioympäristössä. Rajapintakoodi voi siten ohjata datan suoraan simulaatiomallille. Data päätyy lopulta vastaanottavan radion rajapintakoodille, joka tietää kutsua aaltomuotokoodin vastaanottavaa metodia.



Kuva 13. DTL-AIF välisten rajapintojen sekvenssikuva simulaatiossa.

Kontti-lähestymistavan heikkouksista on hyvä olla tietoinen. Jos kyseessä on laajuudeltaan suuri aaltomuoto-ohjelmisto, jokaisen ohjelmistokomponentin välisen yhteyden abstraktoiminen ja rajapinnan hyödyntäminen lisää koodin kompleksisuutta valtavasti. Koodin kompleksisuuden kasvun myötä koodin ylläpidettävyys ja luettavuus vähenevät eksponentiaalisesti. Hankaluuksia voi myös aiheuttaa kontin rajapintojen toteutuksien yhdistäminen simulaatiomallin metodeihin. Jos simulaatioympäristön päivitys muuttaa radikaalisti simulaatiomallin metodeja, koko kontin toiminta voi lakata.

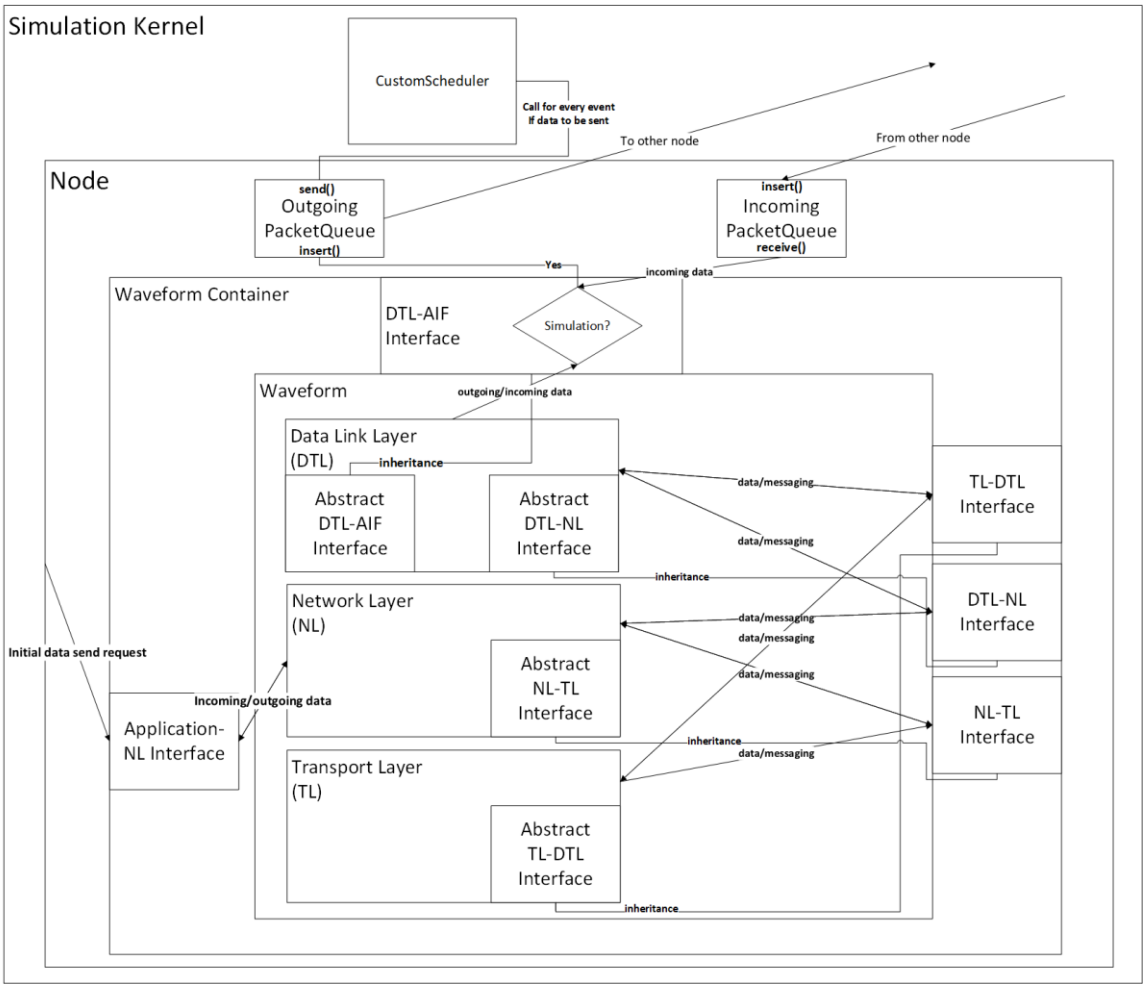
5.3.2 Kaksisuuntainen konttiarkkitehtuuri

Toinen teoreettisistä ohjelmistoarkkitehtuurimalleista on *Two-way Container Architecture* (kaksisuuntainen konttiarkkitehtuuri), joka on suunniteltu tarjoamaan vähemmän riskejä kuin aaltomuotokonttiarkkitehtuuri. Arkkitehtuurimallin keskeinen idea on sama kuin edeltäjälläänkin, mutta se laajentaa samaa konseptia lisäämällä myös simulaatiomallin ympärille kontin. Simulaatiomallin ympärille muodostettavan kontin idea on täysin sama kuin aaltomuodonkin kontin; abstraktoida yhteyksiä ulkopuolisille ja/tai sisäisille komponenteille. Kaksisuuntainen kontti lieventää riskiä siitä, että kontin toiminta lakkaa simulaatioympäristön päivityksen jäljiltä.

Liitteessä 1 havainnollistetaan kaksisuuntaista konttiarkkitehtuuria. Kaikki ohjelmiston osat toimivat simulaation sisällä ja kommunikoivat keskenään abstraktoitujen rajapintojen kautta. Tämä mahdollistaa ohjelmiston tuomisen helposti simulaatioympäristöön, sillä ainoastaan kahden kontin yhteydet tarvitaan liittää toisiinsa. Liitteestä voi myös huomata, mitkä osat konteista on toteutettu simulaatiomoduulien avulla ja mitkä eivät, sillä simulaatiomoduulien liitännät on kuvattu porteilla. Portteja havainnollistavat pyöreäkulmaiset neliöliitännät moduulien välillä. Kaikki Node-moduulin sisällä sijaitsevat paketin lähettämisen askeleet tapahtuvat funktioiden avulla ilman simulaatiomoduulien hyödyntämistä. Jos aaltomuodon sisäiset komponentit viestivät keskenään, niiden liitännät voivat olla abstraktoituja, vaikka eivät johtaisikaan aaltomuodon ulkopuolelle. Täten aaltomuodon toimintaa ei tarvitse suunnitella ympäristöstä riippuvaksi.

5.4 Teoreettinen aaltomuotoarkkitehtuuri

Tämä kappale käsittelee sitä, miltä teoreettinen aaltomuotoarkkitehtuuri, joka olisi suunniteltu ottaen huomioon dokumentissa aiemmin käsitellyt epäyhteensopivuudet, näyttäisi. Havainnollistamisen selvyyden vuoksi arkkitehtuuri on suunniteltu aaltomuoto-kontti-arkkitehtuurin pohjalta. Teoreettisen aaltomuodon ydinrakenne koostuu kolmesta OSI-mallin mukaisesta ohjelmistokerroksesta: DTL (Data Link Layer), NL (Network Layer) ja TL (Transport Layer). Tämän lisäksi, ohjelmiston ydinrakenne sisältää kontin rajapintatoteutusta varten abstraktit rajapintaluokat. Näistä rajapintaluokista periyttämällä saadaan luotua toiminnan sisältävät rajapintaluokat. Kuva 14 sisältää yllä mainitut ohjelmiston elementit.



Kuva 14. Teoreettisen aaltomuodon arkkitehtuurikuva simulaatiossa.

Kuvassa 14 on näkyvissä useita eri tasoja simuloitavan prosessin rakenteesta. Kaikki logiikka on ajossa simulaatiokernelin sisällä. Node-moduuli on yhdistelmämoduuli, joka sisältää aaltomuodon kontin ympäröimänä, lähtevien pakettien jonon (*OutgoingPacketQueue*) sekä tulevien pakettien jonon (*IncomingPacketQueue)*. Aaltomuotokontti koostuu C++-rajapintaolioista, jotka ovat periytyneet aaltomuodon kerrosten tarjoamista abstrakteista rajapintaluokista sekä itse aaltomuodon ydintoteutuksesta. Periytyneet rajapintaluokat löytyvät kuvasta aaltomuodon ympäriltä, kuten konttiarkkitehtuuri nimi kuvastaakin. Aaltomuoto hyödyntää rajapintaluokkien tarjoamia metodeja viestintään muiden komponenttien kanssa. Jokaiselta aaltomuodon kerrokselta on havainnollistuksen takia yksi rajapinta muille kerroksille, mutta todellisuudessa mikään ei rajoita näiden rajapintojen määrää. Jos toteutukseltaan erilaiset rajapinnat ovat tarpeellisia, mikään ei estä uusien rajapintojen luontia. Tältä osin konttiohjelmistoarkkitehtuuri on erityisen joustava, sillä rajana yhteyksien määrässä onkin käytännössä vain ohjelmiston kompleksisuus.

Simulaatiossa aaltomuodon käyttö on kuvattu myös kuvassa 14. Simulaatiomallin tulee lähettää aaltomuodolle pyyntö lähettää dataa toiselle aaltomuodolle. Selkeyden vuoksi toinen Node-moduuli ei ole näkyvissä kuvassa 14, mutta sen oletetaan olevan identtinen kuvattuun Node-moduuliin verrattaessa rakenteellisesti. Node-moduulin metodista lähetetään pyyntö aaltomuodolle kontin *Application-NL_Interface* (Sovellus-Network-kerros rajapinta) kautta lähettää dataa toiselle Node-moduulille. Teoreettisen aaltomuodon ylin kerros on Network-kerros, joten se vastaanottaa viestin ensimmäisenä *Application-NL_Interface*-rajapinnalta. Koska tämä opinnäytetyö ei käsittele aaltomuodon ydintoimintaa, oletetaan, että Network-kerroksen operaatioiden jälkeen paketti siirtyy Transport-kerrokselle. Network-kerros kutsuu sen tiedossa olevan *NL-TL_Interface* (Network-Transport kerrosten rajapinta) olion metodia, jolloin paketti siirtyy *NL-TL_Interface*-rajapinnan toimesta Transport-kerrokselle. Transport-kerroksen jälkeen, samaa periaatetta käyttäen paketti siirtyy Data Link-kerrokselle. Data Link-kerrokselta paketti siirtyy *DTL-AIF_Interface* (Data Link - Air Interface rajapinta) olion avulla Node-moduulin lähtevien pakettien jonolle. *DTL-AIF_Interface* osaa ohjata paketin simulaatiolle, koska se on tietoinen siitä, että ohjelmistoa ajetaan simulaatiossa.

Lähtevien pakettien jonossa paketit odottavat niin kauan, kunnes simulaatio siirtyy seuraavan tapahtuman käsittelyyn. *CustomScheduler* (muokattu järjestelijä) on periytetty OMNeT++:n järjestelijästä ja siihen on lisätty toiminnallisuutta, joka tarkistaa onko aaltomuodoilla paketteja odottamassa lähetystä. Järjestelijä kutsuu lähtevien pakettien jonon lähetysmetodia, johtaen paketit määränpäihinsä. Vastaanottavalla Node-moduulilla paketti ohjataan Node-moduulin toimesta saapuvien pakettien jonoon. Jonosta paketti ohjautuu taas *DTL-AIF_Interface* olion vastaanottometodille. Täten aaltomuodot voivat simulaatiossa viestiä keskenään.

5.5 Muut ratkaisut

OMNeT++:n kehittäjät ovat myös kiinnostuneet siitä, miten heidän simulaatioympäristöönsä voisi tuoda valmiita simuloitavia ohjelmistoja. Ongelman ratkaiseminen tarkoittaisi käyttäjämäärän kasvua heidän ohjelmistolleen. He ovat tunnistaneet, että ohjelmistot ovat monimutkaistuneet runsaasti viimeisen vuosikymmenen aikana. Tämän vuoksi simulaatiomallin muodostaminen ohjelmistoille on hankalampaa kuin aiemmin.

OMNeT++:n kehittäjät ehdottavat artikkelissaan, että uuden simulaatiomallin alusta asti tuottamisen sijaan kehittäjät voisivat tuoda valmiita implementaatioita ohjelmistoista simulaatioympäristöön [34]. Tämä lähestymistapakaan ei kuitenkaan ole ongelmaton, sillä sen mukana seuraa koodin ylläpidon ongelmat sekä yhdistettyjen ohjelmistojen, eli simulaatiomallin sekä simuloitavan ohjelmiston, ylläpitoon liittyvät ongelmat.

Yksi ehdotettu ratkaisu on ulkoisten kirjastojen lataus useaan kertaan. Jos simuloitava ohjelmisto voidaan kääntää kirjastoksi, se voidaan koettaa ladata ajonaikaisesti useaan kertaan. Pienillä muokkauksilla kirjaston lataamisprosessiin, jokaiselle kirjaston instanssille saataisiin oma virtuaalinen muistitila, joten singleton-menetelmää käyttävät ohjelmistokomponentit eivät aiheuttaisi ongelmia. OMNeT++:n kehittäjät totesivat, että tämä ratkaisu ei skaalaudu hyvin satoihin tai tuhansiin instansseihin kirjastoista, sillä käyttöjärjestelmän asettamat rajoitukset tulevat vastaan ensin. [34.] Keinona kirjaston moneen kertaan lataaminen ajonaikaisesti ei ole paras vaihtoehto, ja keino itsekin perustuu siihen, että käyttöjärjestelmän metodeja käytetään niitten suunnitellun käytön ulkopuolella.

OMNeT++:n kehittäjät esittelevät useita muita teoreettisia keinoja ongelman ratkaisemiseksi mutta harva niistä on sellainen, mikä ei vaatisi koodin muokkausta aina kun uusi versio ohjelmistosta halutaan tuoda simulaatioympäristöön. Muokkausta vaativat keinot ovat valitettavan työläitä ja ne saattavat aiheuttaa ongelmia, jos simuloitavan ohjelmiston rakennetta halutaan muuttaa kehityksen aikana tai uutta versiota kehittäessä. Valitettavasti vaikuttaa siltä, että edes OMNeT++:n kehittäjät eivät osaa sanoa yhtä varmaa ratkaisua kyseiseen ongelmaan.

6 Yhteenveto

Työssä todettiin, että ohjelmiston tuominen simulaatioympäristöön on vaativampaa kuin miltä se voi vaikuttaa. Simuloitavan ohjelmiston rakenteen mukaan haasteiden vaativuustaso voi suurestikin vaihdella. Aaltomuodon ja OMNeT++:n tapauksessa ohjelmistoarkkitehtuurien yhteensopivuus ei ollut mahdollista epäyhteensopivuuksien haastavuuden vuoksi. Singleton-menetelmän poistaminen ohjelmiston rakenteesta olisi yksinomaan vienyt runsaasti aikaa ja vaivaa, joita tämän työn tutkimuksen tulosten avulla yritetään välttää.

OMNeT++-simulaatiomallin toteutus tulisi suunnitella samanaikaisesti tai jopa ennen itse simuloitavan ohjelmiston ohjelmistoarkkitehtuuria. Kun simulaatiomallin rakenne on tiedossa, sen asettamat rajoitukset voidaan ottaa huomioon simuloitavassa ohjelmistossakin. Jos simuloitava ohjelmisto on toteutettu konttiohjelmistoarkkitehtuurimallilla, sen tarjoamat rajapintayhteydet on mahdollista suunnitella suoraan simulaatiomalliin kytkettäviksi etukäteen, helpottaen itse ohjelmiston toteutusta. Pitää kuitenkin ottaa huomioon, kuinka kompleksinen simuloitavan ohjelmiston ydintoiminta on. Erittäin kompleksisille ohjelmistoille konttiohjelmistoarkkitehtuurin hyödyntäminen voi johtaa ongelmiin, kun todetaan ohjelmiston laajuuden räjähtäneen käsiin.

Aaltomuodon ulkoisten ja sisäisten liitosten rakennetta tulee miettiä tarkkaan, jos aaltomuotoa kaavaillaan ajettavaksi usealle eri alustalle. Erilaisten yhteysmetodien hyödyntäminen on vaivattomampaa konttiohjelmistoarkkitehtuuria hyödyntävässä ohjelmistossa. Suunnitteluvaiheessa on kannattavaa ottaa huomioon myös muita erityisiä tekijöitä kuin niitä, joita tässä dokumentissa käsiteltiin. Jokainen ohjelmisto on yksilöllinen, eikä tämän dokumentin löydöksiä tule pitää yleismaailmallisena ohjeena jokaiseen ohjelmistoon. On kuitenkin toivottavaa, että dokumentin löydösten yleinen muoto on selvinnyt lukijalle, jotta tieto siitä minkälaisia ongelmia simulaatioympäristöön ohjelmiston tuominen voi aiheuttaa on.

Lähteet

1 OMNeT++ Manual, Parameters. Saatavilla https://doc.omnetpp.org/omnetpp/manual/#sec:ned-lang:parameters. Haettu 2/16/2021, 2021.

2 OMNeT++ Manual, The NED Language. Saatavilla https://doc.omnetpp.org/omnetpp/manual/#cha:ned-lang. Haettu 2/16/2021, 2021.

3 Kenington P. RF and Baseband Techniques for Software Defined Radio. Norwood: Artech House; 2005.

4 Waveform definition. Saatavilla https://techterms.com/definition/waveform. Haettu 3/28/2021, 2021.

5 Youngblood G. A Software Defined Radio for the Masses, Part 1. Saatavilla https://www.arrl.org/files/file/Technology/tis/info/pdf/020708qex013.pdf. Haettu 3/28/2021, 2021.

6 What is software defined radio used for? Saatavilla https://www.onesdr.com/2020/01/15/what-is-software-defined-radio-used-for/. Haettu 3/28/2021, 2021.

7 Choi BK, Kang D, Choi BK. Modeling and Simulation of Discrete Event Systems. Somerset: John Wiley & Sons, Incorporated; 2013.

8 What is OMNeT++? Saatavilla https://omnetpp.org/intro. Haettu 2/11/2021, 2021.

9 GitHub releases for repository omnetpp. Saatavilla https://github.com/omnetpp/omnetpp/releases?after=omnetpp-2.0p1. Haettu 2/13/2021, 2021.

10 Simulation Software. Saatavilla https://sourceforge.net/software/simulation/. Haettu 2/13/2021, 2021.

11 Nutaro JJ. Building Software for Simulation : Theory and Algorithms, with Applications in C++. Hoboken: John Wiley & Sons, Incorporated; 2010.

12 OMNeT++ Manual, Defining Simple Modules. Saatavilla https://doc.omnetpp.org/omnetpp/manual/#sec:simple-modules:defining-simple-modules. Haettu 2/17/2021, 2021.

13 OMNeT++ Frontpage. Saatavilla https://omnetpp.org. Haettu 1/20/2021, 2021.

14 OMNEST Licensing FAQ. Saatavilla https://omnest.com/licensingfaq.php. Haettu 1/6/2021, 2021.

15 OMNEST Licensing Details. Saatavilla https://omnest.com/comparison.php. Haettu 1/6/2021, 2021.

16 OMNeT++ Architecture Paper. Saatavilla https://doc.omnetpp.org/publications/1416290.pdf. Haettu 1/19/2021, 2021.

17 What is INET Framework? Saatavilla https://inet.omnetpp.org/Introduction.html. Haettu 1/13/2021, 2021.

18 INET Showcases. Saatavilla https://inet.omnetpp.org/docs/showcases/. Haettu 1/13/2021, 2021.

19 OMNeT++ Architecture Paper, The Design of OMNeT. Saatavilla https://doc.omnetpp.org/publications/1416290.pdf. Haettu 2/2/2021, 2021.

20 OMNeT++ Architecture Paper, Separation of Model and Experiments. Saatavilla https://doc.omnetpp.org/publications/1416290.pdf. Haettu 2/2/2021, 2021.

21 Matloff N. Introduction to Discrete-Event Simulation and the SimPy Language. Saatavilla http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf. Haettu 2/13/2021, 2021.

22 MIL-HDBK-232A. Military Handbook Red/Black Engineering-Installation Guidelines. Saatavilla https://www.wbdg.org/FFC/NAVFAC/DMMHNAV/hdbk232a.pdf. Haettu 3/26/2021, 2021.

23 Bittium Tough SDR Handheld™. Saatavilla https://www.bittium.com/tactical-communications/bittium-tough-sdr-handheld#application-note. Haettu 3/21/2021, 2021.

24 LUOTTAMUKSELLINEN

25 LUOTTAMUKSELLINEN

26 Nasonov A. The Singleton In C++ - A Force For Good? Saatavilla https://accu.org/journals/overload/14/76/nasonov_1328/. Haettu 2/16/2021, 2021.

27 Singleton example in C++. Saatavilla https://gist.github.com/pazdera/1098119. Haettu 2/16/2021, 2021.

28 Thread-Safe Initialization of a Singleton. Saatavilla https://www.modernes-cpp.com/index.php/thread-safe-initialization-of-a-singleton. Haettu 2/18/2021, 2021.

29 LUOTTAMUKSELLINEN

30 LUOTTAMUKSELLINEN

31 LUOTTAMUKSELLINEN

32 OMNeT++ Manual, Chapter 3.7 Gates. Saatavilla https://doc.omnetpp.org/omnetpp/manual/#sec:ned-lang:gates. Haettu 3/5/2021, 2021.

33 OMNeT++ Manual, Chapter 19.4.21 Inheritance. Saatavilla https://doc.omnetpp.org/omnetpp/manual/#sec:ned-ref:inheritance. Haettu 3/5/2021, 2021.

34 Porting Real-World Protocol Implementations into OMNeT++. Saatavilla https://docs.omnetpp.org/articles/porting-code-into-omnetpp/. Haettu 2/19/2021, 2021.

35 Threads in omnet, simultaneous executions in OMNeT++. Saatavilla https://groups.google.com/g/omnetpp/c/35qmEmrhzB4. Haettu 2/10/2021, 2021.

36 Milea A. Dynamic Memory Allocation and Virtual Memory. Saatavilla https://www.cprogramming.com/tutorial/virtual_memory_and_heaps.html. Haettu 2/23/2021, 2021.

37 Storage class specifiers, Static local variables. Saatavilla https://en.cppreference.com/w/cpp/language/storage_duration. Haettu 2/26/2021, 2021.

38 Labella T, Dietrich I, Dressler F. BARAKA: A Hybrid Simulator of SANETs. Saatavilla https://www.researchgate.net/publication/4259658_BARAKA_A_Hybrid_Simulator_of_SANETs. Haettu 1/3/2021, 2021.

39 Goralski W. The Illustrated Network : How TCP/IP Works in a Modern Network. San Francisco: Elsevier Science & Technology; 2017.

Liitteet

1 Kaksisuuntainen konttiarkkitehtuurikuva.