jamk.fi

IETF tiedonsiirtostandardien kehittäjänä

Jukka-Pekka Mäkelä

Opinnäytetyö
11 2013

Ohjelmistotekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

KUVAILULEHTI

<table>
<tr><td rowspan="3">Tekijä
Mäkelä, Jukka-Pekka</td><td>Julkaisun laji
Opinnäytetyö</td><td>Päivämäärä
23.11.2013</td></tr>
<tr><td>Sivumäärä
84</td><td>Julkaisun kieli
Suomi</td></tr>
<tr><td></td><td>Verkkojulkaisulupa myönnetty
(X)</td></tr>
<tr><td colspan="3">Työn nimi
IETF tiedonsiirtostandardien kehittäjänä</td></tr>
<tr><td colspan="3">Koulutusohjelma
Ohjelmistotekniikka</td></tr>
<tr><td colspan="3">Työn ohjaaja(t)
Peltomäki, Juha</td></tr>
<tr><td colspan="3">Toimeksiantaja
Relator Oy</td></tr>
<tr><td colspan="3">Tiivistelmä

Tiedonsiirtotekniikan nopea kehitys on vaatinut valtavan määrän työtä. Työtä ovat tehneet eri puolilla maailmaa niin eri yritykset ja yhteisöt, kuin yksilötkin. Kullakin toimijalla on ollut omat tavoitteensa, ja se on johtanut valtavaan määrään erilaisia laite- ja ohjelmistoratkaisuja. Aikaisemmin nämä toisistaan poikkeavat ratkaisut muodostivat rajattuun käyttöön tarkoitettuja, toisensa kanssa yhteensopimattomia tiedonsiirtoverkkoja. Nykyäänkin on tarvetta rakentaa laitteistoltaan varsin erilaisia tietoverkkoja eri käyttötarkoituksiin, mutta tiedon halutaan olevan siirrettävissä verkosta toiseen.

Tiedon siirtäminen erilaisten verkkojen välillä on mahdollista määrittelemällä verkkojen välille yhteisiä rajapintoja. Rajapintojen määrittäminen niin, että eri toimijat haluavat ja kykenevät toteuttamaan niitä, on valtava haaste. Tarvitaan teknisistä yksityiskohdista sopimisen lisäksi taloudellisten ja muiden intressien yhteensovittamista. Opinnäytetyössä keskityttiin tutkimaan IETF-yhteisön toiminta malleja määrittelemällä ja julkaisemalla uusi tiedonsiirtokoodaus osaksi IETF:n ylläpitämää RFC-kirjastoa.

Tiedonsiirtokoodauksen määrittelemiseksi tutkittiin IETF:n julkaisemia suosituksia hyväksi havaituista tiedonesitysmuodoista ja olemassa olevia tiedonsiirtokoodauksia. Tutkimuksen pohjalta koodaukseen valittiin laajasti tunnettuja ratkaisumalleja. Valitut mallit pyrittiin määrittelemään mahdollisimman yksikäsitteisesti ja yksinkertaisesti, IETF:n käytäntöjä noudattaen, koodauksen uudelleenkäytettävyyden varmistamiseksi.</td></tr>
<tr><td colspan="3">Avainsanat (asiasanat)
Tiedonsiirto, tietoliikenneverkot</td></tr>
<tr><td colspan="3">Muut tiedot
Liitteinä Ruoska Encoding määritelmäasiakirja, 24 sivua. Ruoska Encoding ohjelmistokirjaston ohjelmointirajapinnat, 15 sivua. Yksikkötesti esimerkkejä, 2 sivua.</td></tr>
</table>

DESCRIPTION

<table>
<tr><td rowspan="3">Author(s)
Mäkelä, Jukka-Pekka</td><td>Type of publication
Bachelor´s Thesis</td><td>Date
23112013</td></tr>
<tr><td>Pages
84</td><td>Language
Finnish</td></tr>
<tr><td></td><td>Permission for web publication
(X)</td></tr>
<tr><td colspan="3">Title
IETF as communication standard developer</td></tr>
<tr><td colspan="3">Degree Programme
Software Engineering</td></tr>
<tr><td colspan="3">Tutor(s)
Peltomäki, Juha</td></tr>
<tr><td colspan="3">Assigned by
Relator Oy</td></tr>
<tr><td colspan="3">Abstract

Development of communication network technologies have required a huge amount of work. Work has been done in different parts of the world, in many different companies and organizations, as well as by individuals. Each actor has their own goals, and it has resulted in a huge amount of different hardware and software solutions. In the past, these divergent solutions formed the networks which were incompatible with each other. Even today, there is a need to build networks with various hardware; however the information is to be moved from network to network.

The transfer of information between different networks is possible by defining common interfaces. Creating network interfaces, so that the different actors are willing to and able to implement them, is a huge challenge. In addition to technical details, economic and other interests have to be matched. This work has been successfully performed by IETF community. The thesis focuses on the study of the practices used by IETF, by defining and publishing new encoding format into the RFC library maintained by IETF.

IETF recommendations and existing encoding formats are studied to find unambiguous definitions for data types and structures for the new encoding format.</td></tr>
<tr><td colspan="3">Keywords
Telecommunication networks</td></tr>
<tr><td colspan="3">Miscellaneous
Attachments: Ruoska Encoding definition document, 24 pages. Ruoska Encoding software library APIs, 15 pages. The unit test examples, 2 pages.</td></tr>
</table>

Sisältö

1 Termit ja lyhenteet....3

2 Työn lähtökohdat....7

3 Taustatutkimus....10

3.1 Yleistä taustatutkimuksesta....10
3.2 Internet Engineering Task Force....11
3.3 RFC-kirjasto....12
3.4 OSI-malli....13
3.5 Koodaustavat....15
3.5.1 Binäärimuotoiset koodaukset....15
3.5.2 Merkkipohjaiset koodaukset....17
3.6 Tietotyypit....19
3.6.1 Totuusarvo....19
3.6.2 Kokonaisluku....20
3.6.3 Merkki ja merkistöt....22
3.6.4 Liukuluku....23
3.6.5 Aikaleimat....24
3.7 Tietorakenteet....25
3.7.1 Taulukot ja merkkijonot....25
3.7.2 Assosiatiiviset taulukot....26
3.7.3 Tietueet....26
3.7.4 Puutietorakenne....27

4 Ruoska Encoding....28

4.1 Yleiset tavoitteet....28
4.2 Koodauksen rakenne....29
4.3 Valitut tietotyypit....29
4.4 Valitut tietorakenteet....31
4.5 Laajennettavuus....32

5 RFC-julkaisuprosessi....33

5.1 RFC-asiakirjan laatiminen....35
5.2 Asiakirjan lähettäminen....36
5.3 Palaute....37
5.4 Julkaisuprosessin lopputulos....37

6 Ruoska Encoding -ohjelmistokirjastototeutus......................................38

6.1 Ohjelmointirajapinnat...39

6.2 Yksikkötestit...40

6.3 Erilaiset ohjelmointiympäristöt...41

7 Pohdinta...42

1 Termit ja lyhenteet

API	Ohjelmointirajapinta (Application Programming Interface) määrittää, miten ohjelmistokomponentit voivat kytkeytyä toisiinsa.
ARM	Advanced RISC Machine on erityisesti sulautetuissa järjestelmissä paljon käytetty prosessoriarkkitehtuuri.
ASCII	Katso US-ASCII.
ASN.1	Abstract Syntax Notation One on standardi, joka sisältää säännöt tietorakenteiden abstraktiin kuvaamiseen ja koodaamiseen ympäristöriippumattomaanmuotoon.
BCD	Binary Encoded Decimal on joukko koodaustapoja, joilla voidaan koodata desimaalilukuja binäärimuotoon.
BCP	Best current practice on IETF:n julkaisema asiakirjasarja. Asiakirjoissa esitellään tämän hetken parhaiksi tunnettuja käytäntöjä liittyen johonkin toistaiseksi standardisoimattomaan tekniikkaan.
BER	Basic Encoding Rules on yksi ASN.1:n binäärimuotoinen koodaus.
CSV	Comma-separated Values on tekstimuotoinen tiedostoformaatti. Se on järjestetty taulukoksi ennalta määritellyillä rivinvaihto- ja sarakkeenvaihtomerkeillä.
FPU	Floating-point unit on erityisesti liukulukulaskentaan suunniteltu suoritin.
GCC	GNU Compiler Collection on GNU projektin tuottama kääntäjäjärjestelmä, joka tukee useita ohjelmointikieliä.
IAB	Internet Architecture Board on ISOC:n toimielin, joka vastaa mm. IETF:n kehittämien standardien yhteensopivuudesta internetin kokonaisarkkitehtuurin kanssa.

IANA	Internet Assigned Numbers Authority on yksityinen yhdysvaltalainen voittoa tuottamaton yritys, joka hallinnoi IP-osoitteita ja muita Internet Protocol -protokollaperheeseen liittyviä numeroita ja symboleita.
IEC	International Electrotechnical Commission on voittoa tuottamaton kansainvälinen standardointiorganisaatio, joka tuottaa ja julkaisee sähkö- ja elektroniikkateknologiaan liittyviä standardeja.
IESG	Internet Engineering Steering Group on IETF:n työryhmäjohtajista ja puheenjohtajasta koostuva ryhmä, jonka vastuulla on IETF:n juoksevien asioiden hoitaminen ja lopullisen teknisen katselmoinnin tekeminen IETF:n työryhmissä kehitetyille standardeille.
IETF	Internet Engineering Task Force kehittää ja mainostaa Internet Standard -standardeja yhteistyössä muiden standardointiorganisaatioiden kuten W3C, ISO ja IEC:n kanssa.
ISO	International Organization for Standardization on Sveitsissä päämajaansa pitävä kansallisten standardisointiorganisaatioiden yhteenliittymä.
ISOC	Internet Society on voittoa tuottamaton kansainvälinen järjestö, joka pyrkii edistämään internetin standardointia, koulutusta ja käyttöä kaikkialla maailmassa.
JSON	JavaScript Object Notation on tekstimuotoinen JavaScript-ohjelmointikielen olioiden kuvauskieli, jota käytetään tiedonsiirtokoodauksena.
NTP	Network Time Protocol on verkkoprotokolla tietokoneiden kellonaikojen synkronoimiseksi tietoverkkojen avulla.

OSI	Open Standards Interconnection on abstrakteihin kerroksiin jaettu malli, jolla pyritään helpottamaan tiedonsiirtoa erilaisten järjestelmien välillä.
Protokolla	Katso tiedonsiirtoprotokolla.
RFC	Request for Comments on IETF:n julkaisema asiakirjasarja.
SDCC	Small Device C Compiler on uudelleen kohdistettava pienille mikroprosessoreille suunniteltu ANSI C -kääntäjä.
SDXF	Structured Data Exchange Format on Max Wildgruben kehittämä tiedonsiirtokoodaus. Julkaistu RFC numerolla 3072.
Tiedonsiirtoprotokolla	Tiedonsiirtoprotokolla on yksityiskohtainen määritelmä viestien rakenteesta ja järjestyksestä, jolla voidaan siirtää tietoa järjestelmästä toiseen.
Unicode	Teollisuusstandardi maailman luonnollisten kielten esittämiseen ja käsittelemiseen tietokoneilla.
US-ASCII	Vuonna 1968 määritelty merkistö, joka sisältää englannin kielen aakkoset, roomalaiset numerot, symboleita ja ohjausmerkkejä. Merkistöön viitataan monesti pelkästään ASCII-nimellä.
UTF-8	Unicoden yksi koodausmuoto. Katso Unicode.
W3C	World Wide Web Consortium on kansainvälinen World Wide Web -teknologioihin keskittyvä standardisointiorganisaatio.
XER	XML Encoding Rules on ASN.1:n XML koodausta tuottava koodaustapa.

XDF	Extensible Data Format on NASA:n kehittämä XML koodattu moniulotteisen tieteellisen tiedon siirtämiseen ja tallentamiseen tarkoitettu formaatti.
XML	Extensible Markup Language on avoin standardi rakenteellisesta kuvauskielestä.
ZACWire	Yhdellä johtimella toimiva kaksisuuntainen tiedonsiirtoprotokolla.
Zigbee	IEEE 802.15 standardiin pohjautuva määritelmä tiedonsiirtämiseksi vähävirtaisissa radioverkoissa.

2 Työn lähtökohdat

Tietoverkot ovat nykyään kaikkialla läsnä ja vaikuttavat jokapäiväiseen elämäämme. Niiden avulla siirretään erilaista tietoa kuten kuvia ulkoavaruudesta maahan, uutisia maailman ääreστä toiseen tai vaikka musiikkia matkapuhelimesta kotistereoihin. Verkkojen moninaiset käyttökohteet ja niiden kehittämiseen osallistuneiden yksilöiden ja organisaatioiden suuri määrä vuosikymmenten aikana on johtanut suureen määrään erilaisia tiedonsiirtoteknologioita, jotka tarvitsevat toimiakseen määrätynlaiset laitteistot ja ohjelmistot. Usein käytännönsovellukset, kuten jäljempänä kuvattu lämpömittarin etäluku, asettavat tarpeen siirtää tietoa järjestelmästä toiseen, ja silloin törmätään laitteistojen ja ohjelmistojen yhteensopivuusongelmiin.

Etäluettava lämpömittari on yksinkertainen esimerkki tiedon siirtämisestä hyvin erilaisten järjestelmien ja niitä yhdistävien erilaisten tiedonsiirtoteknologioiden avulla tietoa tuottavan langattoman lämpösensorin ja käyttäjän matkapuhelimen välillä. Järjestelmässä on digitaalinen lämpötilasensori, joka julkaisee tiedon ZACWire-sarjaliikenneväylällä. Väylässä on myös pieni akkukäyttöinen radiolaite, joka lukee lämpötilan väylältä ja lähettää sen Zigbee-radioverkkoon. Zigbee-verkossa on puolestaan yhdyskäytävälaite, joka kuuntelee lämpötilatiedon radioteitse ja välittää sen edelleen USB-väylän avulla sulautetulle tietokoneelle. Tietokone välittää mittarilukeman internetyhteyden avulla palvelimelle. Internet käsitellään tässä esimerkissä yhtenä kokonaisuutena, vaikkakin se todellisuudessa muodostuu suuresta määrästä erilaisia tiedonsiirtoteknologioita, joiden läpikäyntiin ei ole tässä yhteydessä mahdollisuutta. Palvelimella lämpömittarin lukema tallennetaan tietokantaan. Tietokannasta lukema siirtyy internetyhteyden avulla käyttäjän matkapuhelimessa toimivaan sovellukseen, jonka tehtävänä on näyttää se puhelimen ruudulla käyttäjälle.

Kuten esimerkistä huomataan, vaatii jo yksittäisen lämpötilatiedon välittäminen tietoa tuottavasta laitteesta käyttäjälle monenlaisen tiedonsiirtoteknologian käyttöä. Kukin teknologia asettaa tietyt vaatimukset ja rajoitteet siirrettävän tiedon esittämiselle. Tiedonsiirron kannalta esimerkki voidaankin pilkkoa viiteen järjestelmäkokonaisuuteen, joissa kussakin lämpötilatieto esitetään ja

käsitellään omassa laitteisto- ja ohjelmistoriippuvaisessa muodossa.

Ensimmäinen osio on digitaalinen lämpötilasensori, joka kirjoittaa lämpötilan 10-bittisenä binäärilukuna ZACWire-väylälle. Mikäli luku halutaan muuttaa reaaliluvuksi, löytyy sensorin datalehdeltä muuntokaava. Kaava ei noudata mitään yleistä standardia, vaan on kyseiselle sensorityypille ominainen. Sensorissa käytetty lämpötilayksikkö on Celsius.

Toinen osio on Zigbee-radiolaitteet, joiden pohjana on 8-bittinen 8051-suoritinarkkitehtuuri. Radiolaitteessa ajetaan C-kielistä ohjelmaa, joka lukee lämpötilasensorin ilmoittaman lukeman ZACWire-väylältä, muokkaa sen 16-bittiseksi kiinteällä desimaalipisteellä varustetuksi BCD-koodatuksi luvuksi ja lähettää sen radioverkon avulla yhdyskäytävälaitteeseen, joka on puolestaan yhdistetty sulautettuun tietokoneeseen USB-väylän avulla. Lämpötilatieto kirjoitetaan USB-väylälle.

Sulautettu tietokone on kolmas osio. Se lukee lämpötilatiedon USB-väylältä, muokkaa sen UTF-8 -koodatuksi merkkijonoksi, sisällyttää XML-muotoiseen viestiin, lisää viestiin tiedon lämpötilayksiköstä, joka tässä tapauksessa on Celsius, ja lähettää edelleen internetyhteyden avulla palvelimelle. Sulautettu tietokone voi olla toteutettu hyvin monenlaisella suoritinarkkitehtuurilla, tässä esimerkissä se on ARM-pohjainen, käyttöjärjestelmänä Linux ja ohjelmisto on C-kielinen.

Neljäs osio on internetpalvelin, joka vastaanottaa XML-viestin, purkaa sen sisältämän lämpötilatiedon ja muokkaa sen 32-bittiseksi liukuluvuksi. Lisäksi palvelinohjelma lukee viestistä lämpötilayksikön ja muuntaa lämpötilan Kelvineiksi ja tallentaa SQL-tietokantaan. Palvelimen suoritinarkkitehtuuri on usein x86-pohjainen, tässä esimerkissä i686 ja käyttöjärjestelmä on Linux. Tietokanta on MySQL ja lämpötilatietoa käsittelevä ohjelmakoodi on Java-kielinen.

Käyttäjän matkapuhelimessa toimiva sovellus, joka muodostaa viidennen osion, ottaa yhteyden palvelimeen ja pyytää viimeisimmän lämpötilatiedon käyttäjän asettamalla lämpötilayksiköllä. Palvelin noutaa lämpötilatiedon tietokannasta ja muuntaa yksikön Kelvineistä käyttäjän pyytämään, mikä on tässä

esimerkissä Farenheit. Lisäksi palvelin muokkaa tiedon merkkijonoksi, sisällyttää XML-viestiin ja palauttaa viestin vastauksena matkapuhelimelta tulleeseen pyyntöön. Puhelinsovellus näyttää lämpötilan käyttäjälle puhelimen ruudulla. Useimmat puhelimet ovat nykyään ARM-pohjaisia kuten tässäkin esimerkissä. Ohjelmisto on Objective C++-kielinen.

Esimerkin lähempi tarkastelu osoittaa, että eri laitteistot ja niissä ajettavat ohjelmistot muodostavat hyvin erilaisia järjestelmiä, joissa tietoa käsitellään. Tämä johtaa siihen, että tiedonesitysmuotoa joudutaan muuttamaan useampaan kertaan sen siirtyessä sitä tuottavalta laitteelta käyttökohteeseensa. Onnistuneen tiedonsiirron ehdoton vaatimus on, että lähettäjällä ja vastaanottajalla on samanlainen tulkinta siirrettävän tiedon merkityksestä. Lähettäjällä ja vastaanottajalla täytyy siis olla jokin keino sopia yhteisestä tavasta tulkita siirrettävää tietoa. Onnistumisen avaimet ovat siis keinoissa kehittää hyviä sopimuksia ja keinoissa varmistaa sopimuksien noudattaminen. Näitä sopimuksia kutsutaan tiedonsiirtoprotokolliksi.

Työn lähtökohtana oli tutkia, mitä vaiheita sisältyy tiedonsiirtoprotokollan kehittämiseen ja mahdolliseen standardoimiseen määrittelemällä mahdollisimman järjestelmäriippumaton tiedonsiirtoprotokolla, jota voitaisiin käyttää niin pienissä sulautetuissa laitteissa kuin isoissa palvelinkoneissa. Näin esimerkin mukaisessa skenaarioissa riittäisi yksi protokolla, eli yksi tiedonesitysmuoto, mikä helpottaisi paljon vastaavien järjestelmien toteuttamista ja testausta.

Taustatutkimusta tehtäessä kävi ilmi, että täysin uuden tiedonsiirtoprotokollan määrittäminen on työnä aivan liian laaja ja ehdotinkin toimeksiantajana toimivalle Relator Oy:lle, että määrittelisin tiedonsiirtokoodauksen, jota voitaisiin käyttää erilaisten protokollien pohjana. Ehdotus hyväksyttiin ja päätettiin, että työssä keskitytään koodauksen määrittelemiseen ja sen julkaisemiseen RFC-kirjastossa.

3 Taustatutkimus

3.1 Yleistä taustatutkimuksesta

Työn tärkeimpänä tavoitteena oli tutkia tietoverkkotekniikkaan liittyvien asiakirjojen, kuten verkkoprotokollien tai tässä tapauksessa uuden tiedonsiirtokoodausmäärittelyn, julkaisemista IETF-organisaation ylläpitämässä RFC-kirjastossa. Kirjaston avulla voi uuden tekniikan kehittäjä saada työlleen näkyvyyttä ja sitä kautta kommentteja, parannusehdotuksia ja mahdollisesti uusia käyttäjiä, koska RFC-asiakirjojen asema uusien verkkotekniikoiden hautomona on laajasti tunnustettu koko internetyhteisössä. Kirjasto on ollut olemassa jo yli 30 vuotta, jona aikana monista julkaistuista asiakirjoista on muodostunut kaikkialla internetissä käytössä olevia standardeja. Lisäksi vuosikymmenten aikana on muodostunut tarkoin määritetyt vaatimuksen julkaistaville asiakirjoille, niiden sisällölle ja muotoilulle, mikä helpottaa asiakirjoihin tutustumista ja niissä kuvatun tekniikan käyttöönottoa ja standardointia.

Tiedonsiirtokoodauksen määrittämiseksi taustatutkimuksessa tutkittiin olemassa olevia koodauksia ja niiden käyttöä tiedonsiirtoprotokollissa. Erilaisten koodausten suuri määrä on pakottanut keskittymään vain joihinkin yleisimmin käytettyihin. Näiden yleisimmin käytettyjen koodausten löytämisessä on käytetty apuna niin Wikipediasta löytyvää Comparison of data serialization formats -listausta kuin kirjallisuuslähteitä ja hakukoneita.

Valituista koodauksista tutkittiin erityisesti, minkälaisia tietotyyppejä ja -rakenteita niissä on käytössä ja millaisia esitysmuotoja niille on määritelty. Tämän pohjalta on pyritty muodostamaan lista yleisimmin käytetyistä tietotyypeistä ja -rakenteista ja mikä tärkeitä, kuinka niiden esitysmuoto voitaisiin määritellä mahdollisimman yksikäsitteisesti. Lisäksi on tutkittu koodausten rakennetta, ohjelmointirajapintoja ja muita seikkoja, jotka vaikuttavat niiden käytettävyyteen erilaisissa ohjelmointiympäristöissä kuten niukasti resursseja omaavissa sulautetuissa laitteissa.

3.2 Internet Engineering Task Force

IETF syntyi vuoden 1986 alussa, kun joukko Yhdysvaltojen hallituksen rahoittamia tutkijoita aloittivat neljännesvuosittaiset kokoontumiset. Jo samana vuonna mukaan kutsuttiin yritysten edustajia, samalla kun kokoontumiset olivat avoimia kaikille asiasta kiinnostuneille. Alussa kokousten osallistujamäärät laskettiin kymmenissä mutta kasvoi nopeasti niin, että vuonna 1998 osallistujia oli yli 2000. Vuonna 1991 kokousten määrä laskettiin vuositasolla neljästä kolmeen. (DiBona & Ockman 1999, 48.)

IETF ei ole koskaan järjestäytynyt lain edessä oikeushenkilöksi. Näin ollen ei voida yksikäsitteisesti määritellä, ketkä ovat sen jäseniä. Käytännössä jäseniksi lasketaan kaikki, jotka osallistuvat toimintaan joko osallistumalla kokouksiin tai sitten keskusteluun sähköpostilistalla. Vuoden 1997 loppuun kulut peitettiin kokousmaksuilla ja Yhdysvaltojen hallituksen tuella. Tästä eteenpäinkin kuluista on osa peitetty kokousmaksuilla, mutta loput on maksanut Internet Society. (Mts. 48.)

Internet Society on vuonna 1992 perustettu lähinnä tarjoamaan oikeudellinen järjestelmä tukemaan IETF:n nimissä tehtävää internetstandardien kehitystyötä. Tämä oli tarpeen, koska IETF:n luonteesta johtuen julkaistuille standardeille ei ollut osoitettavissa oikeuden edessä vastuullista tahoa, mikä johti siihen, että toiminnassa mukana olevat yksityishenkilöt pelkäsivät, että joutuvat mahdollisesti henkilökohtaisesti vastamaan lain edessä, mikäli standardien käytöstä tai niissä mahdollisesti olevista virheistä aiheutuu ongelmia standardin käyttäjille. Lisäksi Internet Society rahoittaa myös RFC-sihteeristön ja näin ollen omistaa kopiointioikeuden RFC-asiakirjoihin. (Bygrave & Bing 2009, 95.)

Sisäisesti IETF on organisoitunut työryhmiin, jotka muodostavat kahdeksan toimialuetta: Applications, General, Internet, Operations and Management, Routing, Security, Transport ja User Services. Toimialueilla on yksi tai kaksi johtajaa, jotka on valittu kaksivuotisille kausilleen sattumanvaraisesti vapaaehtoisten joukosta. Vapaaehtoiseksi voi ilmoittautua, jos on ollut mukana kahdessa kolmesta viimeisimmästä toimialueen kokouksesta. Alueiden johtajat ja koko IETF:n puheenjohtaja muodostavat IESG-ryhmän. IESG toimii IETF:n

standardien hyväksyntäelimenä. Lisäksi on kaksitoistahenkinen IAB, joka puolestaan toimii IESG:n neuvonantaja työryhmiä muodostettaessa ja arvioi työryhmien ehdotusten vaikutuksia suurempiin arkkitehtonisiin kokonaisuuksiin. IAB:n jäsenet valitaan samalla menetelmällä kuin toimialuejohtajat. (DiBona & Ockman 1999, 49.)

Yksi kantavista ajatuksista, mikä erottaa IETF:n monista muista standardisointiorganisaatioista, on, että toiminta ohjautuu alhaalta ylöspäin. Käytännössä tämä tarkoittaa sitä, että IESG tai IAB puuttuvat hyvin harvoin työryhmien muodostamiseen. Ne syntyvät kun asiasta kiinnostuneet jäsenet kokoontuvat ja ehdottavat ryhmän muodostamista. Tämä puolestaan johtaa siihen, että IETF ei pysty luomaan aikatauluja työtehtävien edistämiseksi, mutta toimintamalli varmistaa, että työryhmissä säilyy innostus ja sitä kautta hyvät onnistumisen mahdollisuudet. (Mts. 51.)

Toinen suuri erottava tekijä IETF:n ja muiden merkittävien standardisointijärjestöjen välillä on avoimuus. Eivät vain kaikki tuotetut standardit ole avoimia, vaan myös kaikki niiden kehitystyöhön liittyvä aineisto on avointa mukaan lukien tapaamiset ja sähköpostikeskustelut. Avoimuus ja mahdollisuus että kuka tahansa voi osallistua kehitystyöhön ovat olleet pääsyyt, miksi standardit ovat menestyneet hyvin. (Mts. 51.)

3.3 RFC-kirjasto

IETF ylläpitää RFC-kirjastoa, jonka lyhenne aikoinaan tarkoitti ”Request for Comments”. Nykyään kaikki julkaistavat asiakirjat ovat käyneet läpi hyvin tarkan katselmoinnin, joten nykyään lyhenteellä viitataan kirjaston sisältämiin asiakirjoihin yleisesti. Kaikista julkaistuista asiakirjoista siirtyy rajoitettu kopiointioikeus IETF:lle. Tämä käytäntö varmistaa, että asiakirja voidaan pitää vapaasti esillä kirjastossa loputtomasti eivätkä alkuperäiset tekijät voi vetää sitä myöhemmin pois. Toiseksi IETF varaa itselleen oikeuden tehdä asiakirjoista johdettuja teoksia, mikäli se on standardisointiprosessissa tarpeen. Muut oikeudet säilyvät alkuperäisillä tekijöillä. (DiBona & Ockman 1999, 50.)

Kirjaston asiakirjat jakaantuvat kahteen pääsarjaa. Ensimmäinen on asiakirjat,

joilla tähdätään standardin muodostamiseen, ja toisen sarjan muodostavat kaikki muut asiakirjat. Standard-sarjan asiakirjat voivat saada prosessissa statuksen Proposed Standard, Draft Standard ja Internet Standard. Muut voidaan jaotella luokkiin Informational, Experimental tai Historic. Lisäksi käytetään väliaikaisasiakirjoja, Internet-Draft, jotka vastaavat parhaiten historiallista "request for comment"-konseptia. Nämä asiakirjat vanhenevat automaattisesti puolessa vuodessa. Näihin luonnoksiin ei saa missään tapauksessa viitata kuin muihin RFC-asiakirjoihin, vaan on tehtävä selväksi, että ne ovat työn alla olevia luonnoksia eivätkä missään nimessä valmiita standardeja. (Mts. 50.)

3.4 OSI-malli

Lämpömittariesimerkissä on käytössä useita tiedonsiirtotekniikoita. Itse anturi on ZACWire-väylällä yhdistettynä Zigbee-radiomoduuliin. Radioverkossa on yhdyskäytävälaite, joka yhdistyy USB-väylän avulla sulautettuun tietokoneeseen. Sulautettu tietokone on kiinteässä yhteydessä internetiin lähiverkon kautta, joka on toteutettu Ethernet-tekniikalla. Kiinteän internetyhteyden tekniikkaa on vaikea määrittää, koska tänä päivänä vaihtoehtoja on hyvin paljon. Sama koskee palvelimen ja matkapuhelimen internet yhteyksiä.

Edellä mainittujen erilaisten verkkolaitteistojen lisäksi järjestelmään kuuluu monenlaisia suoritinarkkitehtuureita kuten Zigbee-moduulin Intel 8051-pohjainen mikrokontrolleri ja sulautetun tietokoneen ARM-pohjainen prosessori. Erityyppisillä suorittimilla ajetaan erilaista ohjelmakoodia.

Ohjelmistojen erojen lisäksi järjestelmässä on käytössä useampia tiedonsiirtokoodauksia. Lämpösensorin ja radiomoduulin välillä ZACWire-väylällä on oma protokolla, radiomoduuli keskustelee yhdyskäytävälaiteen kanssa Zigbee-protokollan avulla ja yhdyskäytävä sulautetun tietokoneen kanssa puolestaan sovellusta varten erikseen kehitetyllä alkeellisella tiedonsiirtoprotokollalla. Sulautetulta tietokoneelta tieto siirtyy XML-koodattuna palvelimelle ja palvelimelta matkapuhelimeen. Tässä välissä se on tallennettuna palvelimella SQL-tietokantaan, jonka rajapinnassa on omat koodauskäytäntönsä.

Kokonaisuutena vaaditaan kaikkien erilaisten laite- ja ohjelmistotekniikoiden

yhteensovittamista, jotta lämpömittariesimerkin mukainen yksinkertainen sovellus olisi mahdollinen. Tietoliikenteessä erilaisten tekniikoiden yhteensovittamista on pyritty helpottamaan ajattelumallilla, jossa monimutkainen kokonaisuus jaetaan kerroksiksi. ISO on kehittänyt kerrosajattelusta OSI-mallin. Malli tarjoaa työkaluja niin laitevalmistajille kuin verkonkäyttäjille määrittää keskenään yhteensopivia rajapintoja, joiden avulla eri tekniikoilla toteutettujen verkkojen yhteensovittaminen olisi helpompaa. (Kaario 2002 18-21.)

Mallin alimpana kerroksena on *fyysinen kerros*, jonka tehtävä on huolehtia bittien siirtämisestä laitteelta toiselle. Bitit voidaan toteuttaa esimerkiksi radiosignaaleina langattomassa verkossa tai sähkösignaaleina johtimissa. (Mts.)

Fyysisen kerroksen päällä on *siirtokerros*, joka hallinnoi fyysistä yhteyttä. Hallinnointiin kuuluu tiedon paketointi fyysisen kerroksen ominaisuuksien mukaan, pakettien lähettäminen, pakettien vastaanotto ja vastaotettujen pakettien oikeellisuuden tarkistaminen. Lisäksi kerros vastaa tarvittaessa siirtotien tarkoituksen mukaisesta jakamisesta useamman sovelluksen kesken. (Mts.)

Seuraavana tulee *verkkokerros*, jonka päätehtävä on tarjota reitityspalvelu. Palvelun avulla paketit löytävät verkon läpi lähteestä kohteeseensa. *Verkkokerroksen* päällä on *kuljetuskerros*, joka alempia kerroksia hyödyntäen tarjoaa ylemmille kerroksille luotettavan tiedonsiirtoyhteyden lähettäjän ja vastaanottajan välille. Kun mahdollisuus yhteyden muodostamiseksi on olemassa, seuraava eli *istuntokerros* määrittää, kuinka yhteys muodostetaan ja puretaan. (Mts.)

Kuudes kerros on *esitystapakerros*, joka on työn kannalta merkittävä, koska työssä määriteltävä koodaus on juurikin kyseisen kerroksen asia. *Esitystapakerroksen* ensimmäinen tehtävä on varmistaa, että lähettävä tieto on muodossa, jonka vastaanottaja pystyy käsittelemään. Lähettävä laite suorittaa tietorakenteiden esitystavan muunnoksen omasta, itselleen luonteen omaisesta muodosta yleisesti tunnettuun siirtomuotoon. Toinen tehtävä on pakata suuret tietorakenteet pienempiin paketteihin tiedonsiirron tehostamiseksi. Kolmas tehtävä on tarvittaessa salata siirrettävä tieto. (Ciccarelli & Faulkner 2004, 20.)

Seitsemäntenä ja ylimpänä kerroksena OSI-mallissa on *sovelluskerros*. Tämä kerros toteuttaa verkkosovelluksien, kuten SMTP tai FTP, protokollat ja tarjoaa rajapinnan varsinaiselle sovellusohjelmille, jotka hyödyntävät näitä protokollia. (Kaario 2002, 21.)

3.5 Koodaustavat

3.5.1 Binäärimuotoiset koodaukset

Tietokoneiden prosessorit kykenevät käsittelemään vain binäärimuotoista tietoa. Tiedosta on siis tehtävä tavalla tai toisella binääriesitys, ennen kuin sitä voidaan prosessoida. Tästä johtuen on ollut myös luonnollista tietotekniikan alkuaikoina tallentaa ja siirtää tietoa binäärimuotoisena. Ei ole tarvittu konversiokerrosta, kun on käytetty prosessorille ominaista esitysmuotoa tiedosta myös sen tallentamiseen ja siirtämiseen. Tämä on tehokas tapa toimia ja on säästänyt paljon tietokoneiden ja ohjelmoijien resursseja. Siinä on vain yksi suuri ongelma. Tietoa voidaan siirtää vain keskenään samanlaisten järjestelmien välillä. Haasteeseen on vastattu luomalla koodauksia, joissa on määritelty säännöt, kuinka tiedot tallennetaan, kun niitä halutaan siirtää tai tallentaa. Ensimmäiset koodaukset ovat olleet hyvin rajoittuneita ja usein tarkoitettu vain jonkin tietyn ongelmakentän ratkaisemiseen. Nopeasti kuitenkin huomattiin, että tiedon esittämiseen tarvittaisiin geneerisempää ratkaisua. Eräs ensimmäisistä edelleen käytössä olevista ratkaisuista on ASN.1.

ISO:n kehittämä ASN.1 on kieli, jolla voidaan kuvata monimutkaisiakin tietorakenteita järjestelmäriippumattomasti. Abstraktin kuvauksen, jolla voidaan kuvata siirrettävän tiedon rakenne, lisäksi ASN.1 sisältää liudan koodaussääntöjä. Sääntöjen avulla siirrettävä tietorakenne muunnetaan lähetettäessä siirrettäviksi biteiksi ja vastaanotettaessa takaisin järjestelmäkohtaiseen muotoon. Tunnetuin näistä säännöstö on BER. (Kaario, 2002 279-280.)

Uusimpia tulokkaita on Googlen kehittämä Protocol Buffers. Suunnittelun lähtökohtana on ollut luoda yksinkertaisempi ja ennen kaikkea tehokkaampi korvaaja XML-koodaukselle. Sisäisesti Google käyttää sitä varsin laajasti tiedon tallentamiseen ja siirtämiseen järjestelmiensä välillä. Sitä on myös mahdollista

käyttää monissa Googlen julkisista ohjelmointirajapinnoista, mikä osaltaan tehostaa tiedonsiirtoa, kun ei tarvitse tehdä muunnoksia, jonkin muun koodauksen, kuten XML:n, ja sisäisesti käytetyn Protocol Buffers-koodauksen välillä. (Developer Guide 2012.)

Protocol Buffers -sisältää kuvauskielen aivan kuten ASN.1. Käyttäjä määrittelee tietorakenteet määrittelytiedostoon. Tiedosto käännetään protoc-kääntäjällä ja näin saadaan tietorakenteen käsittelyyn tarvittava lähdekoodi halutulle ohjelmointikielelle. Tämä helpottaa ohjelmistojen kehitystä havainnollistamalla tietorakenteita ja poistamalla rakenteiden koodaukseen ja purkamiseen tarvittavan ohjelmiston ohjelmoinnin. Google on julkaissut kääntäjästä avoimen lähdekoodin versiot niin C++, Java kuin Python -ohjelmointikielille. Lisäksi löytyy paljon kolmannen osapuolten tekemiä kääntäjiä muille kielille. (Mt.)

Binäärimuotoisen koodauksen etuna pidetään tehokkuutta. Tiedon esitysmuodon muuttaminen siirtomuodosta järjestelmäriippuvaiseen muotoon ja päinvastoin on usein varsin suoraviivainen prosessi. Avoimen tiedonsiirron kannalta heikkoutena on pidetty sitä, että binäärimuotoista tietoa on hyvin vaikea ihmisten tulkita semmoisenaan. Tiedon purkamiseksi tulkittavaan muotoon tarvitaan aina käytännössä ympäristö- ja koodausriippuvainen ohjelmisto. Toisaalta yritykset ovat usein toteuttaneet omia protokolliaan tai tiedostomuotoja nimenomaan binäärimuodossa varjellakseen salaisuuksiaan liittyen tiedon siirto- tai tallennustekniikoihin.

3.5.2 Merkkipohjaiset koodaukset

Merkkipohjaiset koodaukset nojaavat tiettyyn merkistöön, jonka mukaan niiden sisältöä tulee tulkita. Merkistöstä on valittu pieni joukko merkkejä, jotka toimivat ohjausmerkkeinä, ja niiden avulla voidaan määritellä koodatun asiakirjan tai viestin rakenne. Viestin sisältö ei saa missään tapauksessa sisältää merkistöstä valittuja ohjausmerkkejä, koska silloin viestin rakenne muuttuu eikä sisältöä voida palauttaa. Tästä syystä merkkipohjaiset koodaukset eivät yleisesti ottaen voi sisältää binääritietoa kuten jpeg-kuvia tai mp3-ääntä, koska binäärimuotoinen kuva saattaa sisältää tavun, joka vastaa jotakin ohjausmerkkiä ja näin sekoittaa merkkipohjaisen viestin rakenteen, eikä sisältöä voida

enää palauttaa.

Merkkipohjaisten koodausten etuna on pidetty niiden helppoa tulkattavuutta. Usein puhutaan ihmisen luettavista (human readable) koodauksista, koska koodatun viestin tai asiakirjan sisällön pystyy ihminen tulkitsemaan, kunhan tuntee koodauksessa käytetyt ohjausmerkit. Tästä onkin suurta apua tiedonsiirtoprotokollan kehitysvaiheessa, kun yksinkertaisilla työkaluohjelmilla voidaan tarkistaa, että viestit ovat sisällöltään ja rakenteelta juuri sellaisia kuin mitä niiden on oletettukin olevan. Virheiden löytäminen on siis ainakin teoriassa helppoa. Todellisuudessa ongelmien löytäminen menee vaikeaksi, kun tietomäärät kasvavat. Silloin viestien tekstimuotoisuudesta huolimatta tarvitaan työkaluohjelmia, jotta niitä voidaan helpommin selata.

Eräs yksinkertaisimmista laajasti käytössä olevista merkkipohjaisista koodauksista on CSV. Koodauksesta ei ole olemassa yhtä yksikäsitteistä määritelmää, vaan sillä usein viitataan tekstimuotoiseen tietoon, joka on järjestetty riveiksi rivinvaihtomerkillä ja rivit ovat jaettu sarakkeiksi pilkulla(,). On olemassa monia variaatioita, joissa pilkun sijasta käytetään jotain muuta merkkiä. Myös rivinvaihtomerkki saattaa vaihdella tai niitä saattaa olla useita. Sarakkeet voivat sisältää tekstimuotoista tietoa, joka ei saa sisältää sarakkeiden erottelumerkkiä eikä rivinvaihtomerkkiä.

CSV on varsin laajasti käytetty kun taulukkomaista tietoa halutaan siirtää kahden muuten yhteensopimattoman järjestelmän välillä. On kuitenkin huomattava, että koodauksen monimuotoisuudesta johtuen jää täysin käyttäjän vastuulle varmistaa että sekä lähettävällä, että vastaanottavalla järjestelmällä on samanlainen tulkinta ohjausmerkeistä ja tiedon sisällöstä. CSV-koodauksen standardointia on pyritty edistämään RFC 4180 määritelmällä.

Nykyään ehkä tunnetuin merkkipohjainen koodaus on XML, jonka voittokulku alkoi 1990-luvun loppupuolella. Alussa käyttö oli hyvin pienen joukon harraste mutta sai nopeasti julkisuutta ja käyttö laajeni. Nykyään on vaikea kuvitella mitään vakavasti otettavaa IT-alan järjestelmää jossa ei käytettäisi XML:ää edes konfigurointitiedostoformaattina. Käytön myötä myös prosessointi ja hallintatyökalut kehittyivät niin hienostuneiksi, että monet käyttäjät eivät edes tajun-

neet miten paljon pellin alla tapahtui. Vaikkakin XML on varmistanut paikkansa niin viime aikoina on tuotu esiin joitakin heikkouksia, jotka ovat johtaneet parannuksiin mutta myös muiden vaihtoehtojen esittelyyn. (Fawcett, Ayers & Quin, 2012, 41.)

XML on kaikkinensa hyvin laaja konsepti eikä ole mahdollisuutta käydä kaikkia osakokonaisuuksia tässä läpi. Monipuolisuus, mikä on sen suurin valtti, on johtanut myös monimutkaisuuteen ja se on johtanut siihen, että XML on joutunut antamaan monissa sovelluksissa tilaa yksinkertaisemmille koodauksille kuten esimerkiksi JSON:lle.

JSON on kehitetty ECMAScript-ohjelmointikielen olionotaation pohjalta. Se soveltuu hierarkkisen tietorakenteiden esittämiseen. Se on pyritty pitämään mahdollisimman yksinkertaisena ja siinä onkin vain neljä perustietotyyppiä, merkkijono, numero, totuusarvo ja null. Lisäksi on tietorakenteet taulukko ja olio. Alkuperäinen suunnittelija on Douglas Crockford ja määritelmä on julkaistu RFC-asiakirjana RFC 4627. Vaikka JSON-koodauksen juuret ovat JavaScript-kielessä niin se ei ole sidottu pelkästään siihen vaan on toteutettavissa millä tahansa muullakin ohjelmointikielellä. Yksinkertaisuuden ja ilmaisuvoimansa vuoksi sille löytyykin nykyään toteutukset hyvin monenlaisiin ympäristöihin.

JSON:in yhtenä etuna XML:ään verrattuna on myös tietotyypit. XML ei itsessään määrittele tietotyyppejä vaan se jää jonkun ylemmän kerroksen kuten tiedonsiirtoprotokollan tehtäväksi. JSON puolestaan määrittää edes perustietotyyppejä, joka osaltaan yksinkertaistaa sen käyttöönottoa.

Merkkipohjaisten koodausten haittana on pidetty usein suurta resurssien kulutusta. Esimerkiksi kokonaisluku usein ilmoitetaan desimaalinumeroista koostuvana merkkijonona. Tällöin kuluu ainakin isompien numeroiden kohdalla paljon enemmän muistia kuin jos käytettäisiin binäärilukuja. Esimerkiksi 16-bitin eli kahden tavun binääriluku voi sisältää desimaalijärjestelmän luvun 65535 minkä esittämiseen merkkijonona tarvitaan viisi tavua eli 2,5 kertainen määrä binääriesitykseen verrattuna. Muistin lisäksi kuluu suoritinaikaa kun desimaaliluvun sisältävä merkkijono käydään läpi ja siitä muodostetaan suoritinarkkitehtuurin mukainen binääriesitys. Näin joudutaan aina tekemään jos luvuille halu-

taan suorittaa matemaattisia toimenpiteitä kuten yhteenlaskua.

Ongelmallista on myös sijoittaa binäärimuotoisia tiedostoja kuten jpeg-kuvia merkkipohjaisesti koodattuihin asiakirjoihin. Tätä on pyritty ratkaisemaan erilaisilla tavoilla koodata binääritiedosto uudelleen merkeillä, jotka eivät sekoitu merkkipohjaisen koodauksen ohjausmerkkeihin. Yksi yleisesti käytetty on Base64, josta tosin on olemassa useita keskenään yhteensopimattomia versioita. Yhteensopimattomuusongelmia on pyritty ratkaisemaan standardisoimalla koodaus, mistä esimerkkinä on RFC 4648. Lisäkoodausten haittana on, että ne kasvattavat binäärimuotoisen tiedon kokoa ja lisäävät prosessointi tarvetta kun tietoa pakataan ja puretaan.

Toinen vaihtoehto on korvata ohjausmerkit usealla merkillä, mitä on käytetty sähköpostin Internet Message Format:issa, jonka määritelmä löytyy RFC-asiakirjasta RFC 5322. Merkkipohjaiseen moniosaiseen sähköpostiviestiin voidaan sisällyttää binääritiedosto kun se erotetaan muusta viestistä boundary-merkkijonolla, jonka sisältö on valittu niin, että sama sisältö ei esiinny binääridatassa.

3.6 Tietotyypit

3.6.1 Totuusarvo

Totuusarvo on yksinkertaisin perustietotyyppi, se voi saada vain arvot tosi (True) tai epätosi (False), mitkä voidaan esittää yhden bitin arvoilla 1 ja 0. Todellisuudessa useimpien ohjelmointikielien toteutuksessa totuusarvo varaa enemmän muistia kuin yhden bitin. Yleisesti vähintään yhden tavun, joissakin järjestelmissä jopa 4 tavua. Ohjelmointikielten esitysmuotoa ei siis voida suoraan käyttää tiedonsiirrossa erilaisten järjestelmien välillä vaan on määriteltävä erikseen esitysmuoto tiedonsiirtoa varten.

Olemassa olevista tiedonsiirtokoodauksista ei ole löydettävissä yhtenäistä esitysmuotoa eikä myöskään IETF ei ole määritellyt mitään suositeltavaa tapaa toisin kuin esimerkiksi kokonaisluvuille suositellaan tiettyä muotoa.

Binäärimuotoisissa koodauksissa, kuten ASN.1-standardin BER-koodaukses-

sa, totuusarvo on usein esitetty yhden tavun levyisellä etumerkittömällä kokonaisluvulla missä 0 tarkoittaa epätosi ja kaikki muut arvot 1-255 ovat tosi. (Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), 2008, 6)

Vastaavasti merkkipohjaisissa koodauksissa, kuten YAML versiossa 1.2, totuusarvo usein esitetään merkkijonoilla ”true” ja ”false”. (YAML Ain’t Markup Language (YAML™) Version 1.2, 2009.)

3.6.2 Kokonaisluku

Kokonaisluku on varsin yleisesti käytetty perustietotyyppi. Se on yksinkertainen ja tarjoaa hyvän suorituskyvyn, koska se on useimmissa järjestelmissä toteutettavissa primitiivisenä tietotyyppinä. Useissa ohjelmointikielissä käyttäjä voi määritellä kokonaisluvun arvoalueen bittien lukumäärällä ja etumerkin olemassa ololla. Mitä suurempi arvoalue tarvitaan sitä enemmän sen esittämiseen varataan muistista bittejä, jotka muodostavat tavuja.

Myös kokonaislukujen esitysmuoto riippuu käytetystä ohjelmointikielestä ja laitteistosta. Suurin ero liittyy arvoalueeltaan laajempien kokonaislukujen tavujärjestykseen. Osa järjestelmistä käyttää niin kutsuttua big-endian -muotoa mikä tarkoittaa sitä, että eniten merkitsevä tavu tallennetaan ensin eli alhaisimpaan muistiosoitteeseen tai tiedonsiirrossa se lähetetään ensimmäisenä. Vastaavasti käytetään myös little-endian -muotoa missä taas vähiten merkitsevä tavu tallennetaan tai lähetetään ensimmäisenä. Näiden kahden lisäksi on olemassa muitakin tapoja järjestää kokonaislukujen tavuja mutta ne ovat niin harvinaisia nykyään etten niiden läpikäyminen ole tässä yhteydessä mielekästä.

Erilaiset tavujärjestykset ovat aiheuttaneet paljon ongelmia jo tietotekniikan varhaista vaiheista lähtien. Kummallakin niin, big-endian kuin little-endian, muodoilla on omat kannattaja joukkonsa, joilla on liuta erilaisia perusteluita miksi juuri heidän tavujärjestystään tulisi käyttää. Asiasta on siis käyty paljon keskustelua jo vuosikymmeniä eikä loppua näy. (Cohen, 1980.)

Internet Protocol -standardissa on määritelty käytettäväksi big-endian -muotoa. Siihen viitataankin alan kirjallisuudessa verkkotavujärjestyksenä (Network byte order). IETF suosittaakin, että kaikessa tiedonsiirrossa käytettäisiin verkkotavujärjestystä. (Internet Protocol, 1981, 38.)

Tavujärjestyksen lisäksi negatiivisten kokonaislukujen esitysmuoto saattaa vaihdella. Nykyisin on yleisesti käytössä kahden komplementtijärjestelmä vaikka aikaisemmin on käytössä ollut myös etumerkillinen ja yhden komplementtijärjestelmät. Tästä ei olemassa suosituksia, johtuen ehkä kahden komplementti muodon yleisyydestä, mutta tiedonsiirrossa on joka tapauksessa määriteltävä mitä muotoa käytetään.

Binäärimuotoisessa tiedonsiirto koodauksissa on sovittava kokonaislukujen osalta tavujärjestys mikäli käytetään leveämpiä kuin yhden tavun kokonaislukuja. Lisäksi on määriteltävä kuinka negatiiviset kokonaisluvut esitetään.

Merkkipohjaisissa koodauksissa kokonaisluvut esitettään usein desimaalijärjestelmän mukaisesti merkkijonoina kuten ”-123” tai ”707”. Tällöin täytyy määritellä myös mitä merkkejä käytetään desimaalierottimena, etumerkkinä ja potenssimerkkiä mikäli niitä tai jotain muita merkkejä tarvitaan. Kokonaisluvut voidaan koodata myös heksadesimaalijärjestelmän mukaisesti merkkijonoiksi kuten ”0xF8”. Esitysmuotoja voi olla myös muitakin eikä mitään yleispätevää sääntöä ei ole olemassa. Useat merkkipohjaiset koodaukset eivät ota edes kantaa asiaan vaan jättävät sen ylemmän kerroksen päätettäväksi.

3.6.3 Merkki ja merkistöt

Useimmat ohjelmointikielet tarjoavat perustietotyyppinä merkkiä. Merkki on useimmissa tapauksissa sisäiseltä toteutukseltaan kokonaisluku. Tätä kokonaislukua vastaava merkki saadaan käytössä olevasta merkkitaulukosta, merkistöstä, jossa kukin numeroarvo vastaa yhtä merkkiä.

Yksi varhaisimmista merkistöistä on ASCII, viralliselta nimeltään US-ASCII. Se polveutui 1800-luvulla käytetystä Baudot-koodistosta. ASCII sisältää englannin aakkoset, roomalaiset numerot, muita symboleita ja ohjausmerkkejä, yhteensä

128 merkkiä, jotka esitetään Baudot-koodin viiden bitin sijaan seitsemällä bitillä. Hyvin pian internetin käytön laajentuessa huomattiin, että ASCII ei riitä, koska eri puolilla maailmaa tarvittiin erilaisia merkkejä kuten kirjaimia, joita ei löydy englantilaisesta aakkostosta. Organisaatiot kuten laite- ja ohjelmistovalmistajat alkoivat kehittää merkistöjä, jotka hyödyntävät kahdeksannen bitin tavusta, joka ei vielä ollut käytössä ASCII-merkistössä. Syntyneet 8-bittiset merkistöt olivat useimmissa tapauksissa yhteensopiva ASCII-merkistön kanssa alimpien 7-bitin osalta mutta eivät olleet keskenään yhteensopivia. Tämä johti ISO8859-standardin syntymiseen vuonna 1985. Standardi määrittää koko joukon merkistöjä pääpainon ollessa eurooppalaisten kielten merkkien tukemisessa. Edelleen oltiin tilanteessa ettei edes kaikkia eurooppalaisia kieliä voitu kirjoittaa yhdellä merkistöllä vaan oli osattava valita oikea merkistö kunkin kielen kirjainten esittämiseksi. ISO 8859 -standardin lisäksi muualla maailmassa oli kehitetty muitakin 8-bittisiä merkistöjä mm. kaukoidän kielten käyttöön.

Internetin käytön edelleen laajentuessa havahduttiin 8-bittisten merkistöjen riittämättömyyteen kuvata maailmalla tarvittavia asiakirjoja, joissa tarvittaisiin useamman kielen merkistöjä. 1980-lopulla ja 90-luvun alkupuolella kehitettiin joitain 16-bittisiä merkistöjä ja aloitettiin työ yhden yhtenäisen merkistön, joka voisi sisältää kaikki mahdolliset merkit, kehittämiseksi. Ohjelmistoteollisuus aloitti työstämään Unicode-merkistöä samalla kun standardisointiyhteisöt ISO ja IEC aloittivat vastaavan työn keskenään nimellä ISO/IEC 10646. Onneksi nämä tahot löysivät toisensa ja nykyään Unicode ja ISO/IEC 10646-standardi ovat yhteensopivia ja niitä kehitetään yhdessä. Teollisuudessa niihin useimmiten viitataan Unicode nimellä.

Unicode-merkistö suunniteltiin aluksi 16-bittiseksi mutta huomattiin, että sen 65536 merkkiään eivät riitä, kaikkien nykyisin tunnettujen luonnollisten kielien merkkien kuvaamiseen. Unicode-merkistö laajennettiin 21-bittiseksi, mikä tarkoittaa mahdollisuutta sisällyttää merkistöön yli miljoona erilaista merkkiä. Unicode merkistö onkin saanut paljon kannatusta, koska UTF-8 koodaus on täysin yhteensopiva US-ASCII merkistön kanssa alimpien seitsemän bitin osalta.

IETF vaatii, että kaikki internetissä käytettävät tiedonsiirtokoodaukset määrittelevät käyttämänsä merkistöt ja UTF-8 tulee ehdottomasti olla tuettujen mer-

kistöjen listalla. (Alvestrand, 1998, 2)

IANA ylläpitää listaa merkistöistä osoitteessa: http://www.iana.org/assignments/character-sets/character-sets.xml.

3.6.4 Liukuluku

1960- ja 70-luvuilla tietokoneiden valmistajat kehittivät kukin omia tapojaan esittää reaalilukuja liukulukuina. Tämä johti ongelmiin kun samat ohjelmat toimivat eri tavalla eri laitteistoissa. Myöskin liukulukuihin liittyvien poikkeustilanteiden kuten nollalla jakamisen käsittely oli kirjavaa. Tämä johti liukulukujen binäärimuodon standardisoimiseen vuonna 1985. IEEE754-standardi sai nimekkäimmiltä suoritinvalmistajilta hyvän vastaanoton ja se on yleisesti käytössä tämän päivän suorittimissa. (Overton, 1996, 5)

Pienissä sulautetuissa laitteista saattaa edelleen puuttua tuki IEEE754-standardin mukaisille liukuluvuille. Osassa saattaa kyllä olla tuki mutta niissä ei ole matematiikkasuoritinta, FPU:ta vaan laskenta tapahtuu ohjelmallisesti, joten lukujen käsittelyyn kuluu paljon aikaa. Sulautetuissa järjestelmissä onkin yleisesti käytetty binääriskaalattuja kokonaislukuja emuloimaan liukulukuja. Binääriskaalattujen kokonaislukujen aritmetiikka on nopeaa ilman matematiikkasuoritinta mutta niitä käytettäessä on pidettävä huolta että mahdolliset ylivuodot havaitaan ohjelmiston oikean toiminnan varmistamiseksi.

Liukulukujen siirtämisessä järjestelmästä toiseen tulee ottaa huomioon kulloisenkin järjestelmän tapa käsitellä liukulukuja. Tiedonsiirrossa on suositeltavaa käyttää laajasti hyväksyttyä IEEE754-standardia.

3.6.5 Aikaleimat

Monissa sovelluksissa tarvitaan erilaisia tapoja ilmaista aikaa. Toisinaan tarvitaan kalenteriaikoja ja toisinaan erilaisia aikajanoja, jotka ilmoittavat erotuksen kahden kalenteriajan välillä. Kalenteriaikojen välittäminen kansainvälisesti on ollut ongelmallista, koska eri maissa on käytössä varsin erilaisia järjestelmiä. Ensinäkin viikonpäivillä ja kuukausilla on eri nimet. Toisekseen kellonaikoja ilmoitetaan sekä kahdentoista, että kahdenkymmenenneljän tunnin mukaan.

Kolmanneksi aikavyöhykkeiden sekä kesä- ja talviaikojen merkitsemisessä on eroja. Lisäksi joissakin maissa on käytössä, länsimaissa käytetyn gregoriaanisen kalenterin sijasta, muita kalenterijärjestelmiä.

Aikaleimojen monimutkaisuuden vuoksi vain harvat tutkituista koodausmäärittelyistä sisältävät aikaa ilmaisevia tietotyyppejä. Useimmat ovat jättäneet asian koodausta käyttävien ylempien kerrosten huoleksi. ASN.1:ssä on kuitenkin määritelty GeneralizedTime ja UTCTime tietotyypit. BER-koodauksessa GeneralizedTime-tyyppi esitetään UTF-8 -koodattuna merkkijonona: YYYYMMDDHHMMSS.fff, missä YYYY on vuosi, MM kuukausi, DD päivä, HH tunnit, MM minuutit ja SS sekunnit. Viimeinen .fff-kenttä ilmoittaa sekunnin tuhannesosat mutta se ei saa sisältää viimeisenä merkkinä nollaa vaan loppunollat pitää supistaa pois. Mikäli tuhannesosia ei ole niin myös piste supistetaan pois.

IETF on julkaissut Proposed Standard -statuksella RFC 3339 -asiakirjan, jolla pyritään edistämään ISO 8601 -standardin mukaisien päivämäärien ja kellonaikojen esitysmuotojen käyttöä tulevissa Internet Standardeissa. Asiakirjassa pyritään määrittämään yksikäsitteiset säännöt sekaannusten välttämiseksi ja sisältää siksi vain osan ISO 8601 -standardin määrittelemistä esitysmuodoista. (Newman, 2002, 6-8.)

3.7 Tietorakenteet

3.7.1 Taulukot ja merkkijonot

Useimmista ohjelmointikielistä löytyy sisäänrakennettuna tuki taulukoille, joiden alkioihin voidaan sijoittaa ainakin kielen perustietotyyppejä tai sitten käyttäjän esittelemiä tietotyyppejä tai -rakenteita. Usein ne toteutetaan muistialueina, joihin keskenään saman tyyppiset ja kokoiset alkiot tallennetaan vieriviereen. Aina ei kuitenkaan ole näin vaan ohjelmointikielestä ja ympäristöstä riippuen toteutus saattaa vaihdella, joten tiedonsiirrossa taulukon esitysmuoto täytyy määritellä erikseen.

Taulukoita voidaan siirtää, joko ilmoittamalla taulukon pituus ennen taulukkoa,

tai sitten määrittelemällä loppumerkki. D-Bus -protokollassa taulukko on määritelty niin, että sen pituus tavuina kerrotaan ensin neljän tavun levyisellä etumerkittömällä kokonaisluvulla, jonka jälkeen tulee neljä kehyksen täyttötavua ja niitä seuraa taulukon alkiot. Taulukon koko on rajoitettu 2^26 eli 67108864 tavuun. Taulukko voi sisältää eri mittaisia alkioita ja siksi alkioiden määrä saadaan selville vasta kun taulukko on käyty alkio alkiolta läpi. (D-Bus Specification, 2012.)

Teksti tallennetaan tietotekniikassa yleisesti merkkijonoihin, jotka ovat useimmissa tapauksissa toteutettu hyvin samalla tavoin kuin taulukot. Merkkijonojen siirtäminen järjestelmästä toiseen on hyvin samankaltainen prosessi kuin muidenkin taulukoiden paitsi että merkkijonon oikean tulkinnan kannalta sekä lähettäjällä että vastaanottajalla tulee olla sama käsitys käytettävästä merkistöstä.

Useissa merkkipohjaisissa koodauksissa merkkijono alkaa ja päättyy lainausmerkkiin(”). Näin ollen merkkijono ei itsessään voi sisältää lainausmerkkiä, eikä muitakaan ohjausmerkkejä. Joissain tapauksissa ohjausmerkkejä voidaan sisällyttää mikäli niiden eteen sijoitetaan escape-merkki, joka on useimmissa tapauksissa (\).

Tutkituista tiedonsiirtokoodauksista ei ollut löydettävissä yhteistä esitysmuotoa taulukoille. Kaikki koodaukset eivät sitä edes eksplisiittisesti tukeneet vaan se oli toteutettava koodauksen muilla rakenteilla ja siten määriteltävä koodausta käyttävässä tiedonsiirtoprotokollassa. Tästä esimerkkinä voi pitää XML-koodausta, jossa taulukon alkiot voidaan esittää peräkkäisinä elementteinä taulukkoelementin sisällä. Elementtien tulkitsemista taulukoksi ei voi nähdä pelkän koodauksen perusteella vaan tarvitaan ylemmän tason tietoa protokollasta.

3.7.2 Assosiatiiviset taulukot

Usein on tarvetta taulukolle, jonka alkioihin voidaan viitata muuten kuin indeksillä. Tällöin käytetään assosiatiivista taulukkoa, jotka muodostuvat avain-arvo -pareista. Taulukosta voidaan hakea tietoa avaimen perusteella ja se voidaan

järjestää avaimen mukaan mikäli avaimet ovat vertailukelpoisia keskenään. Assosiatiivisten taulukoiden toteutukset ja näin myös esitysmuodot eri ohjelmointiympäristöissä vaihtelevat suuresti. Tämä tarkoittaa, että tiedonsiirrossa on määriteltävä erikseen mitä esitysmuotoa käytetään.

Binäärimuotoisessa MessagePack-koodauksessa assosiatiivinen taulukko esitetään ilmoittamalla alussa yhdellä tavulla taulukon tyyppi. Tyyppi voi olla fixmap, jolloin alkioiden määrä ilmoitetaan tyypin kanssa samassa tavussa käyttäen neljää alinta bittiä. Tyyppi voi olla myös map16 tai map32, jolloin alkioiden määrä ilmoitetaan, joko 16- tai 32-bittisellä, etumerkittömällä kokonaisluvulla tyyppitavun jälkeen. Tyypin ja alkioiden määrän jälkeen tulevat alkiot. Alkio muodostuu kahdesta oliosta, jotka voivat olla mitä tahansa MessagePack-koodauksen tukemaa tyyppiä. Järjestyksessä ensimmäinen olio on alkion avain ja toinen on alkion arvo. (MessagePack specification, 2013.)

3.7.3 Tietueet

Tietue on tietorakenne, joka sisältää määritellyn listan nimettyjä tietojäseniä. Esimerkkinä voidaan pitää vaikka osoitetietuetta, jossa on henkilön nimi, katuosoite, postinumero ja postitoimipaikka omina jäseninään. Useimmissa ohjelmointikielissä jäsenet voivat olla tyypiltään perustietotyyppejä, tietorakenteita, toisia tietueita tai osoittimia, johonkin edellä mainituista. Samoin useimmissa ohjelmointikielten toteutuksissa tietojäsenet tallennetaan muistiin siinä järjestyksessä kuin ne ovat esitelty.

Tietueet voivat muodostaa varsin monimutkaisia tietorakenteita, joiden esitysmuodot vaihtelevat suuresti ympäristöstä toiseen. Tästä syystä ei ole olemassa yleistä standardia kuinka tietueet tulisi tiedonsiirrossa esittää vaan kukin tutkituista koodauksista on toteuttanut tämän hyvin eri tavoilla. Voidaan kuitenkin sanoa että useimmissa binäärimuotoisissa koodauksissa tietueiden jäsenet tunnistetaan niiden järjestyksen mukaan.

Osassa koodauksista ei ole olemassa mitään erityistä rakennetta tietueiden esittämiseen vaan tietueiden muodostaminen koodauksen tukemista perustietotyypeistä on jätetty ylemmän tason tehtäväksi. Toisissa on varsin massiivi-

siakin rakenteita mutta niiden täysimittainen hyödyntäminen vaatii peruskoodaussääntöjen lisäksi kuvauskielen. Kuvauskielen avulla rakenteiden hahmottaminen ja siten määrittäminen helpottuu. Binäärimuotoisista koodauksista sekä ASN.1 että Googlen Protocol Buffers ja vastaavasti merkkipohjaisista XML sisältävät kuvauskielet koodaussääntöjen lisäksi.

3.7.4 Puutietorakenne

Puutietorakenteella voidaan kuvata eri tietoalkioiden riippuvuudet toisiinsa ja näin rakentaa hyvinkin monimutkaisia malleja. Käytännössä tämä voidaan toteuttaa tietueilla, joilla on mahdollisuus viitata toisiinsa. Tämä on siis tietueiden erikoistapaus ja koodaus tapahtuu useissa tapauksissa samalla tavalla kuin tietueilla.

Koodaukset voidaan rakenteensa osalta jakaa karkeasti kahteen ryhmään. Ensimmäisessä ryhmässä koodattu tieto muodostuu listasta koodattuja perustietotyyppejä. Toisessa ryhmässä koodattu tieto puolestaan muodostaa puurakenteen. Kummassakin tapauksessa voidaan puurakenne kylläkin muodostaa, ensimmäisessä ryhmässä se vaatii ylemmän tason tietoa miten listan jäsenet liittyvät toisiinsa muodostaakseen puurakenteen. Tämä voi mahdollisesti vaikeuttaa koodauksen käyttöönottoa, koska tuo ylemmän tason tietoa ei välttämättä ole saatavilla, ainakaan kehitystyön alkuvaiheessa, kaikissa tarvittavissa ympäristöissä.

4 Ruoska Encoding

4.1 Yleiset tavoitteet

Ruoska Encoding, jäljempänä RSK, on binäärimuotoinen tiedonsiirto- ja tallennuskoodaus. Se on syntynyt tarpeesta siirtää tietoa hyvin erilaisten järjestelmien kuten mikrokontrollereihin perustuvien sulautettujen laitteiden, palvelintietokoneiden ja käyttöpäätteiden kuten matkapuhelinten välillä. Tärkeimpinä suunnittelulähtökohtina on ollut tehokkuus ja mahdollisimman yksikäsitteinen ja yksinkertainen rakenne. Näin on pyritty varmistamaan, että koodaus on mahdollista toteuttaa järjestelmissä, joissa on hyvin niukalti käytettävissä re-

sursseja kuten työmuistia.

Samalla kun koodaus on pyritty pitämään mahdollisimman yksinkertaisena on siihen sisällytetty kattava joukko yleisesti käytettyjä tietotyyppejä. Tämä mahdollistaa tehokkaan virheiden jäljittämisen, debuggauksen, koodaustasolla, mikä helpottaa koodaukseen perustuvien protokollien tai tallennusformaattien kehitystyötä. Varsinkin uuden protokollan kehitystyön alkuvaiheessa, kun protokollatason virheiden jäljitystyökalua, debuggeria, ei välttämättä ole olemassa, on tärkeää, että koodaustasolla voidaan suorittaa mahdollisimman kattavaa debuggausta.

Perinteisesti on monissa protokollissa esimerkiksi aikaleimat koodattu 32-bittisinä kokonaislukuina samoin kuin vaikka IPv4 osoitteet. Näin ollen koodaustasolla toimiva debuggeri ei ole voinut tehdä eroa IPv4 osoitteiden ja aikaleimojen välillä vaan se on jäänyt debuggerin käyttäjän vastuulle. Nyt kun aikaleimalla on oma tietotyyppi voi debuggeri näyttää aikaleiman suoraan ihmisen helposti ymmärrettävässä muodossa.

4.2 Koodauksen rakenne

Koodauksen perusyksikkö on kehys. Kehyksiä on kolmenlaisia, metakehyksiä, jotka määrittävät rakenteen ja datakehyksiä, jotka säilövät siirrettävän tiedon kehyksen määrittämässä tietotyypissä. Kolmas kehystyyppi on Extended.

Useimpiin kehyksiin, kaikkiin datakehyksiin, voidaan liittää tunniste. Tunniste voi olla tyypiltään tyhjä, etumerkitön kokonaisluku tai merkkijono. Tunnistetta voidaan käyttää RSK pohjaisissa sovelluksissa yksilöimään tiettyjä kehysinstansseja tai määrittämään koodauksen tietotyypeistä johdettuja sovellustason tietotyyppejä.

Kehykset muodostavat tiukasti määritellyn puutietorakenteen. Koodattu data alkaa aina Begin-kehyksellä ja päättyy End-kehykseen. Näiden kehysten välissä on sisennystaso yksi, joka voi sisältää datakehyksiä tai Null ja Array -kehyksiä puurakenteen lehtinä tai Begin- ja End-kehyspareja puun oksina. Oksat voivat puolestaan sisältää lisää lehtiä ja alioksia.

4.3 Valitut tietotyypit

Datakehysten tietotyypeiksi valikoitui pitkän pohdinnan ja taustatutkimuksen pohjalta joukko mahdollisimman yksikäsitteisesti määriteltyjä tietotyyppejä. Tyyppien esitysmuodot on valittu tarkkaan IETF:n suositusten mukaisesti, niiltä osin kuin suosituksia on ollut olemassa.

Totuusarvo on omana tietotyyppinään ja se kuvataan yhdellä bitillä yhden tavun kehyksen sisällä. Tämä on kaikkein optimaalisin tapa, koska yhden tavun kehys on koodauksen pienin yksikkö. Mikäli tarvitaan suuri määrä totuusarvoja voi RSK pohjaiset sovellukset esitellä bittikarttatyypin, kokonaislukutyyppien pohjalta, jossa voidaan säilöä useita totuusarvoja tavussa.

Kokonaisluvuiksi valikoitui 8-, 16-, 32- ja 64-bittiä leveät etumerkilliset ja merkittömät kokonaislukutyypit. Tavujärjestykseksi oli helppo valita big-endian, koska se on saanut teollisuudessa verkkotavujärjestyksen aseman ja sitä suositellaan käytettäväksi internetin tiedonsiirrossa. Negatiivisten lukujen esitysmuodosta ei löytynyt varsinaista suositusta mutta kahden komplementti -muoto on niin yleisesti käytössä, että se tuli valituksi.

Aluksi pohdittiin järjestelmää, jossa kokonaislukujen tavujärjestys olisi mahdollista määrittää jokaisen koodatun asiakirjan tai kehyksen alussa, koska näin saavutettaisiin muutamia etuja. Edut tosin rajoittuvat erikoistapauksiin, kuten siihen, että tiedonsiirron kummankin osapuolen käyttämä tavujärjestys on sama ja se on ennalta tiedossa niin toteutuksia voitaisiin yksinkertaistaa ja nopeuttaa määrittelemällä myös tiedonsiirron tavujärjestys samaksi. Tässä tapauksessa ei kumpaankaan osapuoleen tarvitsisi toteuttaa ohjelmakoodia mikä tarkastaisi tavujärjestystä ja tarvittaessa kääntäisi sitä järjestelmän omasta verkkotavujärjestykseen tai päinvastoin. Haittana tästä olisi osapuolten sisäisen toteutuksen heijastuminen tiedonsiirtoon ja sitä kautta hankaloittaa järjestelmän muuttamista ja laajentamista.

Reaalilukutyypiksi oli luontevaa valita IEEE754-standardin mukainen esitysmuoto, joka on yleisesti käytössä ja monesta järjestelmästä löytyy siihen hyvä tuki. Reaalilukutyypit sisältävät 16-, 32- ja 64-bittiä leveät muodot. Standardi ei

määritä tavujärjestystä mutta kuten kokonaisluvuissa myös reaaliluvuissa käytetään verkkotavujärjestystä.

Merkkijono on toteutettu niin että sen pituus kerrotaan ensin ja merkistöksi on valittu UTF-8, koska se on IETF:n suositusten mukainen. Mikäli jotain muuta merkistöä välttämättä tarvitaan tulee merkkijono sijoittaa Binary-kehykseen ja käytetty merkistö määritellä RSK-pohjaisessa sovelluksessa. Tämä ei ole kuitenkaan suositeltavaa sillä koodaustasolla ei ole tämän jälkeen enää ymmärrystä kuinka tietoa tulisi tulkita. Parempi tapa on sovellustasolla muutta merkkijonot UTF-8 -muotoon.

Aikaleimoja varten on varattu useampi tietotyyppi. RFC 3339 pohjalta on valittu kolme kalenteriaikaleimaa. Ne ovat UTF-8 -koodattuja merkkijonoja, joilla voidaan kertoa joko päivämäärä, päivämäärä ja kellonaika UTC-aikavyöhykkeellä sekunnin tai millisekunnin tarkkuudella. Nämä riittävät moneen sovellukseen ja ISO 8601 -standardi mahdollistaa, että monet järjestelmät tunnistavat aikaleimat ja voivat käsitellä niitä.

Merkkipohjaiset aikaleimat ovat kuitenkin paljon resursseja, kuten muistia ja suoritinaikaa, kuluttavia eivätkä siksi sovellu käytettäviksi järjestelmissä, joissa resursseja on niukalti tarjolla. Siksi RSK tarjoaa niiden lisäksi RFC 5905 -asiakirjassa määritellyt binäärimuotoiset NTP-aikaleimat. Näillä on vahva asema NTP-palvelun yleisyyden myötä. Lisäksi on määritelty täysin uusi RSK Date -aikaleima, joka on optimoitu versio NTP Date -aikaleimasta. Erona on, että RSK Date käyttää vain 8-bittiä aikakauden esittämiseen toisin kuin NTP Date 32-bittiä. Lisäksi RSK Date käyttää vain 32-bittiä sekunnin osien esittämiseen ja NTP Date 64-bittiä.

Sovellustasolla voi olla tarpeita tietotyypeille, joita on mahdoton muodostaa koodaustason tietotyypeillä. Tällöin on mahdollista käyttää Binary-kehystä, johon voidaan sijoittaa mielivaltaista dataa. Se voi olla esimerkiksi kuvaa tai ääntä. Kehyksen sisällön tulkinta ei kuulu koodaukseen vaan se jää RSK pohjaisen sovelluksen asiaksi.

4.4 Valitut tietorakenteet

RSK-koodattu tieto muodostaa aina puutietorakenteen, jolla on yksi juurielementti ja tarvittava määrä oksia ja lehtiä. Tietueet voidaan esittää oksina, joiden lehdet ovat tietueen tietojäseniä. Tietojäsenten ja lehtien yhteys voidaan määrittää joko niiden järjestyksellä tai asettamalla lehtiin tunnisteet. Tunnisteiden käyttö lisää siirrettävän tiedonmäärää ja näin heikentää tiedonsiirron tehokkuutta. Vastapainona saadaan joustavampi järjestelmä ja helpotetaan vikojen jäljittämistä.

Puurakenne voi lehtinään sisältää taulukoita. Taulukoihin voi sisällyttää mitä vain tietotyyppejä, joilla on painolastitavuja ja joille on mahdollista antaa tunniste. Taulukot voivat olla myös assosiatiivisia niin että avaimena toimii taulukon alkion tunniste ja arvona alkio. Array-kehyksestä ilmenee alkioiden määrän lisäksi kaikille alkioille yhteinen kehys- ja tunnistetyyppi.

Puurakenne tuli valituksi monipuolisuutensa vuoksi. Taulukot voidaan toteuttaa Array-kehyksellä tai listana oksia tai lehtiä. Array-kehys on tehokkain tapa, koska silloin taulukon alkioiden ja niiden mahdollisten tunnisteiden tyyppi voidaan määrittää yhteisesti yhdellä tavulla Array-kehyksessä sen sijaan että tämä määritys tehtäisiin alkiokohtaisesti. Monissa sovelluksissa taulukot sisältävät tietueita ja silloin niitä ei yleensä voida suoraan sijoittaa Array-kehykseen. Tällöin on käytännöllisempää käyttää listaa oksista. Lisäksi joissain sovelluksissa ei ole järkevää siirtää isoja taulukoita yhdessä kehyksessä. Syynä voi olla esimerkiksi se, että niiden käsittely vaatisi enemmän muistia kuin on tarjolla tai sitten taulukon kokoa ei voida määritellä etukäteen. Näissä tapauksissa on järkevää käyttää oksista tai lehdistä muodostuvia listoja.

4.5 Laajennettavuus

Useimmissa tapauksissa tiedonsiirtotekniikoihin, aivan kuten muihinkin teknisiin ratkaisuihin, kohdistuu käyttöönoton jälkeen muutospaineita. Koodaukseen onkin sisällytetty jatkokehityksen kannalta tärkeä Extended-kehystyyppi. Tämä mahdollistaa myöhemmin uusien kehystyyppien esittelemisen. Vanhat ohjelmistokirjastototeutukset eivät tosin osaa tulkita uusia kehystyyppejä ilman

päivitystä.

Laajennettavuudessa pyrittiin aluksi taaksepäin yhteensopivuuteen. Ajatuksena oli määrittää Extended-kehyksille yhteinen rakenne pituus kentän osalta. Tämä mahdollistaisi vanhan version mukaisten ohjelmistokirjastototeutusten hypätä Extended-kehysten yli vaikka ne eivät niiden sisältöä osaisikaan tulkita. Yleiskäyttöisen mutta silti tehokkaan pituuskentän määrittäminen osoittautui kuitenkin liian ongelmalliseksi ja jätettiin pois koodauksen määrittelystä. Tämähän ei tarkoita sitä, etteikö RSK pohjaisten protokollien eri versiot voisi olla taakse- tai eteenpäin yhteensopivia. Ainoa ehto on, että kaikki protokollaversiot käyttävät samaa versiota RSK-koodauksesta.

5 RFC-julkaisuprosessi

Kuten taustatutkimuksessa kävi ilmi on RFC-asiakirjojen julkaisemisella pitkät perinteet ja prosessi on muotoutunut pitkän ajan saatossa nykyiseen muotoonsa. Julkaisuprosessista on kirjoitettu paljon, ja se on saanut erilaisia muotoja eri käyttötarkoituksiin. RFC-kirjasto pitääkin sisällään asiakirjoja, joissa eri prosessit ja niiden vaiheet ovat kuvattuina.

Tutkimus alkoi BCP-luokassa vuonna 1996 julkaistusta RFC-asiakirjasta numerolla 2026 ja otsikolla The Internet Standards Process -- Revision 3. Asiakirja kuvailee prosessin internet standardien luomiseksi ja otetaan kantaa syntyvien standardien immateriaalioikeuksiin. Asiakirja korvaa aikaisemmin numerolla 1602 julkaistun RFC-asiakirjan ja sitä itseään on päivitetty julkaisun jälkeen useilla muilla RFC-asiakirjoilla. Tämä on yksi RFC-kirjaston piirre, asiakirjoja ei muuteta julkaisun jälkeen vaan mahdolliset korjaukset, tarkennukset ja päivitykset julkaistaan omina asiakirjoinaan, joihin lisätään viittaukset mitä aikaisempia ne mahdollisesti täydentävät tai korvaavat. Vanhat jätetään aina voimaan niiden alkuperäisillä numeroillaan, jotta viittaukset niihin pysyisivät ehjinä.

Prosessi kuvaa kahden erilaisen polun. Toinen on protokollille, jotka tähtäävät avoimen standardin asemaan ja niitä nimetään Internet Draft ja toinen on RFC vaikkakin kaikille asiakirjoille annetaan RFC numero ja ne julkaistaan RFC-kir-

jastossa. Tämän päätason jaon lisäksi asiakirjat jaetaan vielä useampaan luokkaan. Luokat ovat Informational, Experimental, StandardsTrack.

Toimeksiantajani sen paremmin kuin minäkään ei ole minkään IETF-työryhmän jäsen, joten ainoaksi mahdollisuudeksi jäi RFC 4846 kuvatun mukainen, Itsenäinen toimitus (Independent Submission) -prosessi. Tässä prosessissa käsitellään ja julkaistaan muiden kuin IETF-työryhmien lähettämiä asiakirjoja. Prosessi rajaa suoraan pois vaihtoehdon, että asiakirja voisi olla mukana IETF-standardin kehityksessä. Jäljelle jää luokitukset Experimental ja Informational, joista valituksi tuli Experimental, joka parhaiten kuvaa työn kokeellista luonnetta. Tämä ei kuitenkaan poissulje mahdollisuutta etteikö asiakirjaa voitaisi myöhemmin julkaista standardiin tähtäävänä, kun ensin asialle on saatu julkisuutta Experimental-luokassa.

IETF tunnustaa itsenäisen toimituksen kautta tulevien asiakirjojen tärkeyden juurikin keskustelun avaamiseksi asioista, jotka ei välttämättä ole kyseisellä hetkellä IETF-työryhmien työlistalla. Näin esimerkiksi akateemisen tutkimuksen tuloksia voidaan nostaa keskusteluun IETF-yhteisössä. Samalla voidaan jakaa kokemuksia olemassa olevista tekniikoista tai laite- ja ohjelmistotuottaja voivat esitellä omia protokolliaan. Lisäksi tämä toimii reittinä kokousmuistioille ja mahdolliselle kritiikille mikä kohdistuu mahdollisesti IETF:n omiin prosesseihin. (Klensin & Thaler, 2007, 3-4.)

Mikäli tähdättäisiin standardin luomiseen tulisi osallistua IETF-työryhmätyöskentelyyn. Työn tulokset julkaistaisiin RFC-kirjastossa Internet-Draft asiakirjana. Ennen julkaisua työryhmän johtajan tehtävänä on kerätä niin sähköpostilistan kuin muiden keinojen avulla ryhmän jäsenten mielipiteitä ehdotuksesta. Mikäli työryhmässä on muodostunut konsensus niin ehdotus annetaan arvioitavaksi IESG-ryhmälle. Arvioinnin ensimmäinen askel on lähettää ehdotus arvioitavaksi koko yhteisölle. Tämän jälkeen yhteisön jäsenillä on kaksi viikkoa aikaa antaa omia arvioitaan ehdotuksesta. IESG kokoaa palautteen ja päättää sen pohjalta viedäänkö ehdotus RFC-sihteerille julkaistavaksi vai palautettaanko se takaisin työryhmälle työstettäväksi.

Mikäli standardiin liittyy epävarmuustekijöitä niin se voidaan julkaista RFC-kir-

jastossa Proposed Standard -statuksella. Tässä tilassa standardiehdotusta pidetään vähintään kuusi kuukautta, jonka aikana on pystyttävä poistamaan epävarmuudet testauksella tai muilla keinoin. Tämä jälkeen ehdotuksen status muutetaan Draft Standard -tilaan. Nyt on viimeistään esitettävä todisteet siitä, että ehdotus ei loukkaa kenenkään immateriaalioikeuksia. Lisäksi ehdotuksen kaikki ominaisuudet tulee olla toteutettuna useampaan erilaiseen ympäristöön. Mikäli joitain ominaisuuksia ei saada toteutettua niin ne on poistettava ehdotuksesta, mikä saattaa johtaa standardin yksinkertaistumiseen.

Aikaisintaan neljän kuukauden päästä Draft-Standard ehdotus voidaan siirtää Internet Standard tilaan. Tämän jälkeen alkuperäisillä tekijöillä ei ole enää oikeuksia muuttaa sitä, vaan kaikki muutokset ovat IETF:n käsissä.

Internet Standard on vakiintunut, monin erilaisin toteutuksin toimivaksi todettu, laajasti hyväksytty ja hyödylliseksi todettu määritelmä, jonkun tietyn ongelman ratkaisemiseksi.(Bradner, 1996, 2.)

Kokonaisuudessaan prosessissa näkyy IETF alhaalta ylöspäin ohjautuvuus. Standardiehdotukset nousevat suoraan jäseniltä eikä niitä määrätä ylhäältä. Myöskin demokraattinen äänestyskäytäntö on hylätty, pelkkä enemmistö ei riitä vaan asioiden etenemiseksi vaaditaan kaikkien asiasta kiinnostuneiden kesken konsensus. Konsensus taas perustuu standardiehdokkainen kokeilemisesta käytännössä saatuihin tuloksiin. David Clark kiteyttikin vuonna 1992 prosessin motoksi:

> ”We reject: kings, presidents and voting.
> We believe in: rough consensus and running code.”

5.1 RFC-asiakirjan laatiminen

RFC-asiakirjan laatimisesta löytyy internetistä paljon aineistoa, sitä julkaisee niin IETF kuin monet yliopistot. Historiallisista ja käytännön syistä yleisesti käytetään ASCII tekstiformaattia. Helpottaa Unix-maailmassa käytettyjen työkalujen kuten grep:in käyttöä. Myös asiakirjojen tai niiden osien kopioiminen esimerkiksi sähköposteihin on helppoa. Toinen vaihtoehto on PostScript-muo-

to mutta sitä suositellaan vain jos on tarvetta monimutkaisille kuvioille.

Asiakirjan muotoilu on määritelty RFC 2223 -asiakirjassa, jota on päivitetty RFC 5741 ja RFC 6949 -asiakirjoilla. Niissä annetaan tarkat ohjeet kuinka sivulla saa olla korkeitaan 57 riviä ja rivillä 72 merkkiä. Alapalkit eivät ole sallittuja eikä asiakirja saa sisältää ylimääräisiä välilyöntejä tai rivinvaihtoja. Yläpalkin sisältö ja muotoilu ovat tarkoin määriteltyjä. Kappaleet tulee olla eroteltu yhdellä tyhjällä rivillä. Asiakirjan sisäiset viittaukset toteutetaan kappaleiden numeroilla eikä sivunumeroilla, koska sivutus saattaa muuttua prosessissa.

Muotoilun lisäksi asiakirjan sisältö ja rakenne on myös määritelty. Asiakirjan ensimmäisen sivun yläpalkista tulee selvitä sen tekijät, otsikko, luokka, status ja lisäksi asiakirjojen numerot, joita kyseinen asiakirja täydentää tai korvaa. Seuraavaksi tulee asiakirjan otsikko ja sitä seuraa korkeintaan muutaman kappaleen tiivistelmä. Tiivistelmän jälkeen on oma osionsa, jossa kuvataan asiakirjan status ja tavoiteltu status. Tämän jälkeen tulevat kopiointioikeuslausekkeet.

Seuraavaksi tulee sisällysluettelo ja tämän jälkeen johdantoluku. Johdannon jälkeen alkaa varsinainen leipäteksti jaoteltuna tarkoituksenmukaisesti lukuihin ja kappaleisiin. Asiakirja lopussa tulee olla oma lukunsa turvallisuus näkökulmien käsittelemiseksi. Mikäli on esitelty tekniikan tarvitsee lisätä jotakin IANA rekisteriin on siitä kirjoitettava oma lukunsa. Lopuksi tulevat viiteluettelo ja tekijöiden osoitetiedot. Viiteluettelon muotoilu on myös tarkoin määritelty.

Tarkkojen määrittelyjen vuoksi asiakirjan kirjoittaminen pelkällä tekstieditorilla on varsin työlästä. Onneksi on kehitetty työkaluohjelmia, kuten xml2rfc, jotka osaavat tehdä tekstinmuotoilun pääosin automaattisesti. Lisäksi xml2rfc osaa lisätä, joitakin pakollisia osioita kuten kopiointioikeuslausekkeet, kunhan sille kerrotaan minkä määritelmän mukaan ne halutaan tehtäväksi. Ruoska Encoding määrittely on kokonaisuudessaan kirjoitettu XML-muotoisena asiakirjana, josta on xml2rfc-työkalulla generoitu vaadittu tekstimuotoinen asiakirja. XML-lähdekoodi on nähtävissä sivulla: http://datatracker.ietf.org/doc/draft-ruoska-encoding/.

5.2 Asiakirjan lähettäminen

Kun kyseessä on yksittäisen henkilön tekemä määritelmäluonnos tulee se lähettää Individual Submission -työkalun avulla, jonka käyttöohjeet löytyvät osoitteesta http://www.rfc-editor.org/indsubs.html.

Itse lähettäminen on suoraviivainen toimenpide. Selaimella valitaan lomakkeeseen teksti- ja XML- muotoiset asiakirjat, lisätään oma nimi ja sähköpostiosoite ja painetaan lähetyspainiketta. Tämän jälkeen IETF lähettää sähköpostin missä on linkki lähetyksen oikeellisuuden tarkastamiseksi ja hyväksymiseksi. Kun linkki on avattu ja lähetys kuitattu siirtyy asiakirja RFC-sihteerin työlistalle.

5.3 Palaute

Ensimmäinen palaute asiakirjan lähettämisestä tuli RFC-sihteeriltä, jonka työlistalle asiakirja päätyi. Hän on Auckland yliopistossa tietojenkäsittelytieteen apulaisprofessori Nevil Brownlee.

Palautetta tuli myös IETF sähköpostilistalta, Bill McQuillan huomautti muutamista kirjoitusvirheistä ja nosti esiin negatiivisten kokonaislukujen esitysmuodon. Aloinkin asiaa tutkimaan ja se johti siihen että määrittelin negatiiviset kokonaisluvut kahden komplementti -muotoon.

Sain myös palautetta toimeksiantajaltani ja ohjaavalta opettajaltani Juha Peltomäeltä. Juha ihmetteli XML-esimerkin mielekkyyttä asiakirjassa ja päädyinkin poistamaan sen turhana jäänteenä aikaisemmista suunnitelmista.

Tähän asti palaute on ollut myönteistä vaikkakin sitä on kokonaisuutena tullut varsin niukasti. Jatkossa sitä tulee toivottavasti lisää kun RSK-koodaus saa lopullisen muotonsa ja sille tehdään toteutuksia useampaan ohjelmointiympäristöön.

5.4 Julkaisuprosessin lopputulos

Toistaiseksi julkaisuprosessi on vielä kesken. Määrittelyasiakirja on nyt lisätty RFC-sihteerin työjonoon ja sen eri versiot, joita kirjoitus hetkellä on seitsemän,

ovat kaikki lähetetty IETF yleiselle postituslistalle kommentoitavaksi.

Prosessin lopputulema riippuu jatkossa lähinnä siitä löytyykö RSK-koodaukselle käyttäjiä, jotka ovat halukkaita viemään asiaa eteenpäin. Mikäli kiinnostus jää vähäiseksi ei koodaus tule saamaan RFC numeroa ja julkaisua. Toisaalta voi olla, että käyttäjiä löytyy ja se julkaistaan ainakin Experimental-luokassa. Prosessin etenemistä voi seurata sivulta: http://datatracker.ietf.org/doc/draft-ruoska-encoding/.

6 Ruoska Encoding -ohjelmistokirjastototeutus

Ohjelmiston toteuttaminen ei ole tämän opinnäytetyön keskiössä mutta, kuten IETF:n periaatteet vaatii, tulee uusilla tekniikoilla olla olemassa toteutukset, ennen kuin niistä voidaan alkaa vakavasti luonnostella standardia, niin on myös tästä tekniikasta luotu ensimmäinen ohjelmistototeutus.

Ruoska Encoding ensimmäinen ohjelmistokirjasto on toteutettu C-ohjelmointikielellä, koska sille on saatavissa kääntäjiä moneen erilaiseen ympäristöön, supertietokoneista aina pieniin mikrokontrollereihin. Kaikki sovellukset eivät tarvitse kaikkia koodauksen tarjoamia ominaisuuksia, mutta tässä on toteutettu kaikki referenssinä myöhemmille tiettyyn sovellukseen optimoiduille toteutuksille. Kirjaston ohjelmointirajapinnat löytyvät liitteistä 2 ja 3.

Toteutuksen ensisijaisena tavoitteena on selkeä referenssinomainen lähdekoodi, joka pyrkii kommunikoimaan koodauksen toimintaa. Tästä syystä optimointiin ei ole kiinnitetty juuri huomiota ja se jääkin sovelluskohtaisille toteutuksille. Ainoa asia mitä on pyritty optimoimaan on työmuistin käyttö, jotta toteutusta olisi mahdollista käyttää myös hyvin rajoittuneissa ympäristöissä. Tämä tarkoittaa joidenkin kehystyyppien käsittelyssä mahdollisesti sitä että suoritinaikaa kuluu enemmän kuin jos käytettäisiin suurempia työmuistipuskureita. Toteutus mahdollistaa puskureiden koon valitsemisen ajonaikaisesti, mikä osaltaan helpottaa kirjaston käyttöä erilaisissa ympäristöissä.

Toteutuksen lähdekoodi on käännösaikaisesti konfiguroitavissa, mikä helpottaa toisen tavoitteen eli ympäristöriippumattomuuden saavuttamista. Käännös-

aikaisesti voidaan esimerkiksi määrittää kuinka isoja kokonais- ja liukulukuja on käytettävissä. Konfigurointi on toteutettu esikääntäjämakroilla, mikä on lähdekoodin selkeyden kannalta huono asia. Tämä on siis ristiriidassa ensisijaisen tavoitteen kanssa mutta ympäristöriippumattomuudessa saavutettu hyöty on suurempi kuin lähdekoodin luettavuuteen aiheutunut haitta.

Lisäksi toteutetukseen kuuluvat yksikkötestit kirjaston keskeisistä toiminnoista ja dokumentaatiot ohjelmointirajapinnoista.

6.1 Ohjelmointirajapinnat

Kirjastototeutus sisältää kaksi ohjelmointirajapintaa. Ensimmäistä käytetään kun tietoa ollaan pakkaamassa RSK-muotoon. Toista käytetään tiedon purkamiseen. Kumpikin rajapinta hyödyntää yksityisesti toteutettua C-tietuetta, jonka jäsenet eivät ole näkyvissä rajapinnan käyttäjälle. Pakkausrajapinnassa tietue on tyypiltään RuoskaEncoder, jäljempänä encoder. Purkurajapinnassa puolestaan käytetään RuoskaDecoder -tyyppiä, jäljempänä decoder. Rajapinnoissa on julkisia funktioita, jotka ottavat parametrikseen osoitteen, joko encoder tai decoder, tietueeseen. Kirjastototeutus sisäisesti muokkaa tietueiden jäseniä sen mukaan mitä julkisia funktioita käyttäjä kutsuu. Vastaava toteutus löytyy esimerkiksi GNU libXml kirjaston Reader-rajapinnasta. (Module xmlreader from libxml2. 2013.)

Pakkaaminen alkaa alustamalla encoder tietue. Alustettaessa määritellään kehysten tunnisteille varattu muistipuskurin maksimi koko, takaisinkutsufunktio ja osoitin käyttäjän alustamaan muistipaikkaan. Encoder kutsuu takaisinkutsufunktiota aina kun sen puskuri on tullut täyteen koodattua dataa. Käyttäjän tulee siis toteuttaa takaisinkutsufunktio niin että sille parametrina tuleva data tallennetaan käyttäjän haluamaan paikkaan. Muistiosoitin, joka annettiin encoderia alustettaessa tulee myös parametrina takaisinkutsufunktiolle ja sitä voidaan käyttää encoder instanssin tunnistamiseen. Kun encoder on alustettu voidaan aloittaa tiedon pakkaaminen.

Pakkausvaiheessa kantavana ajatuksena on, että käyttäjä ensin asettaa käytettävän kehystunnisteen encoderiin, mikäli tunnistetta halutaan käyttää. Tun-

niste liitetään tämän jälkeen kaikkiin pakattaviin kehyksiin kunnes tunnistetieto asetetaan uudestaan tai tyhjennetään.

Purkuvaihe aloitetaan alustamalla decoder tietue. Alustuksessa määritellään vastaavat ominaisuudet kuin encoderin alustuksessa. Nyt takaisinkutsufunktiolla on vain siinä mielessä eri merkitys, että sitä kutsutaan aina kun decoderin muistipuskuri on tyhjentynyt ja se tarvitsee lisää purkamatonta dataa. Käyttäjän toteuttama takaisinkutsufunktion tulee siis lukea käyttäjän määrittelemästä paikasta purkamatonta dataa ja palauttaa se RSK-koodauksen purkajalle. Purkamattoman koodin lähde, jota takaisinkutsufunktio käyttää, voi olla sovelluksesta riippuen muistipuskuri, tiedosto tai vaikka jokin IO-laite.

Purkurajapinta tarjoaa tehokkaat välineet tarkastella minkälainen kehys on dekoodattavissa seuraavaksi. Tarkastelun yhteydessä voidaan lukea kehyksen mahdollinen tunniste. Tarkastelua voidaan suorittaa eri funktioilla moneen kertaan ilman, että tämä kuluttaa purettavaa dataa. Kun tarkastelun avulla on tiedossa seuraavan kehyksen tyyppi niin voidaan purkaa sen sisältämä tieto. Mikäli tarkasteluvaiheessa huomataan, että kehys ja sen sisältö ei syystä tai toisesta kiinnosta niin sen yli voidaan hypätä ja siirtyä seuraavaan kehykseen. On kuitenkin huomattava, että hyppy tapahtuu rekursiivisesti eli jos rajapinnan skip-funktiota kutsutaan puurakenteen oksanhaarassa niin hypätään samalla kertaa koko oksan yli. Vastaavasti taulukon alussa tai sisällä skip-funktion kutsuminen aiheuttaa hyppäämisen kaikkien vielä taulukossa jäljellä olevien alkioiden yli. Mikäli purettavan datan kehysjärjestys on ennalta tunnettu voidaan kehykset ja niiden mahdolliset tunnisteen lukea ilman tarkastelua.

Kummankin rajapinnan funktiot yleisesti ottaen palauttavat totuusarvon tosi aina kun operaatio on onnistuu ja vastaavasti epätosi jos operaatio epäonnistuu. Käyttäjän tulee siis aina tarkastaa paluuarvo.

6.2 Yksikkötestit

Kirjastototeutuksen ja määritystyön tueksi olen hyvin varhaisesta vaiheesta asti tehnyt yksikkötestejä. Testit varmistavat, että kulloinkin toteutettu ominaisuus toimii halutulla tavalla. Toisaalta lyhyillä testeillä on voitu hakea koke-

muksia, joiden avulla koodauksen määrittelyä on voitu viedä eteenpäin.

Testejä syntyikin niin paljon, että niiden hallinnointi alkoi jossain vaiheessa olemaan haastavaa. En kuitenkaan halunnut ottaa käyttöön kolmannen osapuolen testauskehystä, koska semmoisen kääntäminen sulautettuihin järjestelmiin saattaisi aiheuttaa valtavasti lisätöitä. Aluksi kirjoitinkin yksikkötestit yksinkertaisina funktioina, jotka assertoivat mikäli jokin ei toiminut niin kuin testissä oletettiin. Kun aloitin tutkimaan kuinka saan käännettyä kirjaston ja testit 8051-pohjaiselle mikrokontrollerille korvasin assertit makroilla, jotka osasivat tulostaa standardiulostuloon millä rivillä testin suoritus on päättynyt virheen vuoksi. Lisäksi tein makrot, jotka osaavat tulostaa testin nimen ja lopputuloksen niin ikään standardiulostuloon. Mikrokontrollerien ohjelmointiympäristöissä ei ole useinkaan minkäänlaista näyttöä mistä tuloksia voitaisiin tarkastella vaan standardiulostulo ohjataan sarjaporttiin, joka on kiinni tietokoneessa. Tietokoneessa sarjaportista tuleva tieto voidaan tulostaa ruudulle tai tallentaa tiedostoon myöhempää tarkastelua varten.

Ympäristöriippumattomuus johti massiiviseen esikääntäjä makrojen käyttöön, mikä heijastui ylenmääräisenä monimutkaisuutena yksikkötesteihin.

6.3 Erilaiset ohjelmointiympäristöt

RSK on tarkoitettu monenlaisiin ympäristöihin. Joissakin ympäristöissä ei ole esimerkiksi tukea IEEE754-liukuluvuille tai 64-bittiä leveille kokonaisluvuille, joten ne täytyy jättää jo käännösvaiheessa pois. Toki nämäkin tyypit voidaan toteuttaa käyttämällä muita perustietotyyppejä mutta se ei ole enää tämän työn piirissä. Lisäksi joissakin ympäristöissä dynaaminen muistin varaaminen voi olla ongelmallista, jolloin on tultava toimeen staattisilla muistipuskureilla. Staattisesti varatut puskurit eivät sinällään ole mikään ongelma kunhan muistipuskurien koko mitoitetaan oikein. Oikea mitoitus taas riippuu puskurien käytöstä ja käytön määrittelee sovellus. Sovelluksien kuten tiedonsiirtoprotokollien suunnittelussa pitää siis ottaa huomioon ettei yksittäisten kehysten koko kasva liian suureksi mikäli sovelluksen on toimittava hyvin rajoittuneessakin ympäristössä.

Erittäin rajoittuneissa ympäristöissä, kuten pienissä mikrokontrollereissa voidaan joutua tekemään vielä radikaalimpia ratkaisuja. Voi olla, että koodauksen toiminnallisuus on sisällytettävä sovelluskohtaiseen ohjelmakoodiin hyödyntäen staattisia rakenteita. Tämä lähestymistapa ei luonnollisesti ole suositeltavaa, koska tällöin loogisesti eri tasoilla olevien toiminnallisuuksien ylläpitäminen ja kehittäminen on erittäin hankalaa.

7 Pohdinta

Onnistuneen tiedonsiirron ehdoton vaatimus on, että lähettäjällä ja vastaanottajalla on samanlainen käsitys siirrettävän tiedon merkityksestä. Ennen tiedonsiirron aloittamista täytyy lähettäjän ja vastaanottajan sopia, kuinka siirrettävää tietoa tulee tulkita. Nämä sopimukset ja niiden laatu ovat siis ratkaisevassa asemassa kaikessa tiedonsiirrossa. Opinnäytetyön tavoitteena oli tutkia, kuinka IETF on onnistunut luomaan maailmanlaajuisesti hyväksyttyjä sopimuksia, standardeja, ja näin edistämään avoimen tiedonsiirron kehitystä.

Opinnäytetyön tuloksena syntyi ehdotus uudesta tiedonsiirtokoodauksesta, nimeltään Ruoska Encoding. Sen toiminnan ja rakenteen määritteleminen lämpötilasensoriesimerkin avulla nosti esiin monia kysymyksiä, joihin vastausten löytäminen johti laajaan taustatutkimukseen. Oli löydettävä vastauksia niin pieniin yksityiskohtiin kuin laajempiin konsepteihin. Pienistä yksityiskohdista esimerkkinä yksinkertaisimman perustietotyypin, totuusarvon, esitysmuodon määritteleminen. Laajemmista konsepteista puolestaan voisi mainita koodauksen puumaisen rakenteen ja sen monikäyttöisyyden.

Uuden koodauksen määritteleminen on osittain ristiriidassa IETF:n periaatteiden kanssa. IETF suosittaa, että uusien tiedonsiirtoprotokollien tulisi hyödyntää ASN.1 tai XML -koodauksia. Toisaalta uuden koodauksen määritteleminen ja julkaiseminen on avannut näkymiä IETF:n toimintamalleihin ja olemassa oleviin koodauksiin on ollut helpompi pureutua käytännön työn kautta.

Jatkossa tarvitaan vielä paljon työtä Ruoska Encoding -ohjelmistokirjastojen toteuttamiseksi eri ympäristöihin. Onhan määritelmä vasta luonnostasolla, ja nyt tarvitaan käytännön sovelluksia testaamaan, mitkä ominaisuudet ovat tar-

peellisia ja mitkä voidaan jättää pois.

Ruoska Encoding -koodausta tärkeämpi tulos on parempi ymmärrys IETF:n toimintamalleista. IETF on osoittanut, miten tärkeää on täydellinen avoimuus, kun halutaan määrittää maailmanlaajuisia ympäristöriippumattomia standardeja. Vapaaehtoisuus ja alhaalta ylöspäin ohjautuvuus ovat myös merkittäviä tekijöitä, kun halutaan motivoida tekijöitä. Lisäksi IETF:n kehitystyössä aikakäsite on kääntynyt päinvastaiseksi totutusta. Viimeistään päivämääristä on luovuttu ja tilalle on otettu aikaisintaan päivämäärät. Ei määritellä, milloin uusi standardi tulisi olla viimeistään valmis, vaan milloin se aikaisintaan voisi valmistua. Näin varmistetaan, että aikataulupaineiden vuoksi ei jätetä tärkeitä työvaiheita kuten käytännön testausta pois. Seuraavan tutkimustyön kohde voisikin olla, miten näitä käytäntöjä voisi soveltaa yritysmaailmassa.

Lähteet

Alvestrand, H. 1998. Charset Policy. Viitattu 16.11.2013. http://www.ietf.org/rfc/rfc2277.txt

Bygrave, L. & Bing, J. 2009. Internet Governance : Infrastructure and Institutions. New York: Oxford University Press

Ciccarelli, P. & Faulkner, C. 2004. Networking Foundations. Alameda, CA, USA: Sybex, 2004. p 20.)

Cohen, D. 1980. ON HOLY WARS AND A PLEA FOR PEACE. Viitattu 16.11-.2013. ftp://ftp.rfc-editor.org/in-notes/ien/ien137.txt

Comparison of data serialization formats. 2013. Viitattu 16.11.2013. http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats

D-Bus Specification. 2012. Viitattu 16.11.2013. http://dbus.freedesktop.org/doc/dbus-specification

Developer Guide. 2012. Viitattu 16.11.2013. https://developers.google.com/protocol-buffers/docs/overview

DiBona, C. & Ockman, S. 1999. Open Sources. Sebastopol: O'Reilly Media, Inc.

Fawcett, J., Ayers, D. & Quin, L. 2012. Beginning XML, 5th Edition (5th Edition). Somerset: Wiley

Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). 2008. Viitattu 16.11.2013. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.690-200811-I!!PDF-E&type=items

Internet Protocol. 1981. Viitattu 16.11.2013. http://www.ietf.org/rfc/rfc791.txt

Kaario, K. 2002. TCP/IP-verkot. Jyväskylä: Docendo Finland Oy

Overton, M. 1996. Floating Point Representation. Viitattu 16.11.2013. https://www.cs.uiowa.edu/~atkinson/m170.dir/overton.pdf

MessagePack specification. 2013. Viitattu 16.11.2013. https://github.com/msgpack/msgpack/blob/master/spec.md

Module xmlreader from libxml2. 2013. Viitattu 16.11.2013. http://www.xmlsoft.org/html/libxml-xmlreader.html

Newman C. 2002. Date and Time on the Internet: Timestamps. Viitattu 16.11-.2013. http://tools.ietf.org/html/rfc3339

YAML Ain’t Markup Language (YAML™) Version 1.2. 2009. Viitattu 16.11-.2013. http://yaml.org/spec/1.2/spec.html

Liitteet

Liite 1. Ruoska Encoding

Network Working Group   JP. Makela
Internet-Draft   October 12, 2013
Intended status: Experimental
Expires: April 15, 2014

Ruoska Encoding
draft-ruoska-encoding-06

Abstract

This document describes hierarchically structured binary encoding format called Ruoska Encoding (later RSK). The main design goals are minimal resource usage, well defined structure with good selection of widely known data types, and still extendable for future usage.

The main benefit when compared to non binary hierarchically structured formats like XML is simplicity and minimal resource demands. Even basic XML parsing is time and memory consuming operation.

When compared to other binary formats like BER encoding of ASN.1 the main benefit is simplicity. ASN.1 with many different encodings is complex and even simple implementation needs a lot of effort. RSK is also more efficient than BER.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 15, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Document Structure 3
1.1. Endianness 4
1.2. String Encoding 4
2. Frame Definitions 5
2.1. Leading Byte 5
2.2. Meta Frames 6
2.2.1. Null Frame 6
2.2.2. Begin Frame 6
2.2.3. End Frame 7
2.2.4. Array Frame 7
2.3. Data Frames 8
2.3.1. Boolean Frame 9
2.3.2. Integer Frames 9
2.3.3. Float Frames 10
2.3.4. String Frame 10
2.3.5. Binary Frame 11
2.3.6. DateTime Frames 12
2.3.7. NTP Short Frame 13
2.3.8. NTP Timestamp Frame 13
2.3.9. NTP Date Frame 14
2.3.10. RSK Date Frame 14
2.4. Extended Frames 15
3. Identifiers 16
3.1. Identifier Types in Leading Byte 16
3.2. Null Identifier 16
3.3. Integer Identifiers 16
3.4. String Identifier 17
4. Frame Type Table 18
5. Implementation Notes 20
6. Security Considerations 21
7. IANA Considerations 22
8. Normative References 23
Author's Address 24

1. Document Structure

The principal entity of RSK document is frame. Two main classes of frames exist. Meta Frames to define structure and Data Frames to hold actual payload data.

All frames start with Leading Byte which defines frame type and some type depended instructions. Some meta frames and all data frames can be tagged with an identifier. Identifier type is defined in Leading Byte. If identifier exists it is placed right after the Leading Byte. In Data Frames payload comes after identifier. Meta Frames may also have payload or special fields or both. Data type of the payload is defined by frame type and type depended instructions. All frame types are explained in Section 2.

RSK document is structured as finite tree. The tree is rooted to Begin Frame. After the rooting Begin Frame follows data frames as leafs and Begin - End Frame pairs as branches. Branches may contain data frames as leafs and again Begin - End Frame pairs as sub branches. Null and Array Frames are considered as data frames here.

RSK document always ends with End Frame. Use of End Frame is two fold. It is used to return from branch to parent level and terminate the document. So document must always start with Begin Frame and end with End Frame. Root nesting level 0 must not contain any other than rooting Begin and terminating End Frames. Between root Begin and terminating End Frame is nesting level 1. Nesting level 1 may contain data frames or branches or both.

Example Tree Structure

```
0  1  2  Nesting levels
|  |  |
Begin[id:tractor]                              Begin Frame at level 0
|  String[id:manufacturer, value:Valmet]         Leaf at level 1
|  String[id:model, value:33D]                   Leaf at level 1
|  Begin[id:engine]                              Branch at level 1
|  |  String[id:fuel, value:Diesel]                Leaf at level 2
|  |  UInt8[id:horsepower, value:37]               Leaf at level 2
|  End                                           Ending branch
End                                            Terminating at level 0
```

Figure 1: Tree Structure

1.1. Endianness

Big-endian networking byte order is used. Endianness applies to all integer and floating point numbers. This includes payload of any data frames like Integer, Float, and Timestamp and 16-bits wide integer identifier values and also meta data fields like length of payload. Canonical network byte order is fully described in RFC791, Appendix B.

1.2. String Encoding

All strings are UTF-8 encoded. This applies to string identifiers and payload of String, Date, DateTime and DateTimeMillis Frames.

Implementations using any of frame types above or String Identifier or both must be able to validate UTF-8 encoding. On writing phase UTF-8 encoding violation must be handled as error condition. If UTF-8 encoding fails on reading phase warning must be raised and let user decide to continue reading or not. More information about UTF-8 see RFC 3629 [RFC3629].

2. Frame Definitions

As mentioned earlier the principal entity of RSK is frame. This section explains all frame types and type dependent special instructions in detail.

2.1. Leading Byte

All frames start with Leading Byte. Leading Byte determines frame type and type dependent instructions. The most significant bit is reserved for Extended Frames which may be introduced in later versions. See Section 2.4 for details.

Leading Byte is presented as bit array where left-most bit is the most significant bit. MSB 0 bit numbering scheme is used with two exceptions. Left-most bit is reserved for Extended Frame and thus marked as 'X' for all Leading Byte definitions. Also some bits are marked with 'R' meaning that they are reserved for later use and must not be set in this version.

Leading Byte is followed by frame type dependent fields like Identifier or Payload or both. These fields are presented as labeled byte blocks with possible lengths in bytes, kilobytes like 64k, or gigabytes like 4G.

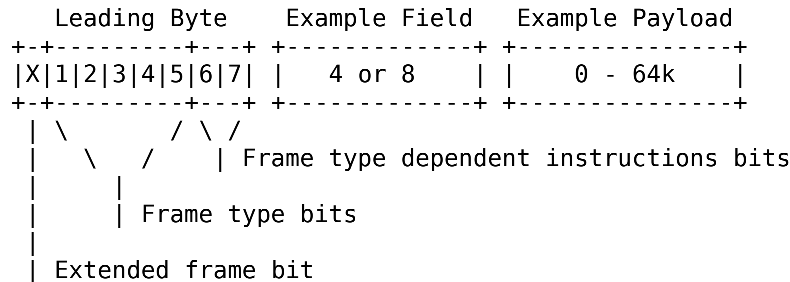

Figure 2: Leading Byte

Example Field: Example of possible frame type dependent byte field. 4 or 8 means that field can be 4 or 8 bytes long. Actual length can be determined by frame type, instruction bits, or some other field.

Example Payload: Example of possible frame type dependent payload field. 0 - 64k means that field can be from 0 to 65535 bytes long. Actual length can be determined by frame type, instruction bits, or some other field.

Frame type dependent instructions bits: These bits determine type dependent instructions. See specific frame type sections for details. For most frame types these are used to define identifier type.

Frame type bits: This bit field determines frame type. Values are defined in Section 4.

Extended frame bit: Extended frame bit. See Section 2.4 for details.

2.2. Meta Frames

Meta Frames define document structure.

2.2.1. Null Frame

Null Frame can be tagged with an identifier but does not contain any payload data.

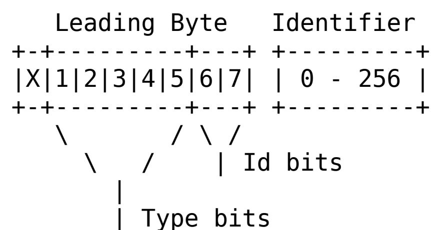

Figure 3: Null Frame

Identifier & Id bits: See Section 3 for details.

Type bits: See Section 4.

2.2.2. Begin Frame

Document and branches start with Begin Frame. Begin Frame may have an identifier. More details about tree structure see Section 1.

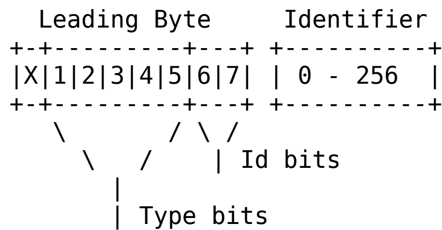

Figure 4: Begin Frame

Identifier & Id bits: See Section 3 for details.

Type bits: See Section 4.

2.2.3. End Frame

End Frame is used to return from branch to parent level in tree structure and also used to terminate a document. More details about tree structure see Section 1.

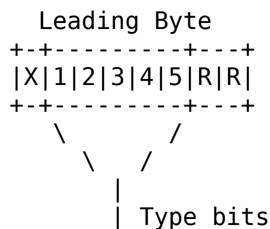

Figure 5: End Frame

Type bits: See Section 4.

2.2.4. Array Frame

Array column in Frame Type Table in Section 4 defines frame types which can be enclosed into a array.

Array itself and each item may have identifiers. Array identifier is defined in Leading Byte. All items have identifier of same type and all items are same frame type. Frame and identifier type for all items are defined by CLB (Common Leading Byte).

Array capacity is defined by selecting corresponding Array Frame

type. See Section 4 for details.

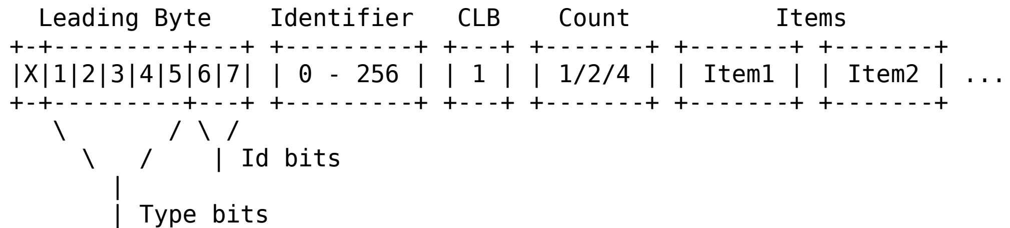

Figure 6: Array Frame

Identifier & Id bits: Array identifier. See Section 3 for details.

CLB: Common Leading Byte determines type of items and type of item identifiers.

Count: 8, 16, or 32-bits wide unsigned integer telling item count. Width of Count field depends on array type, see Section 4 for details.

Items: Array of items.

Type bits: See Section 4.

Array items are stored right after Count field. Items may have Identifier.

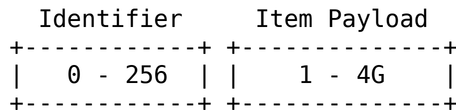

Figure 7: Array Item

2.3. Data Frames

Data Frames are collection of widely used data types. There are frames for boolean, integer and floating point numbers, UTF-8 encoded strings, dates, and timestamps. There is also frame for raw binary data. All data frames can be tagged with an identifier.

2.3.1. Boolean Frame

Boolean value (False/True) is defined by choosing corresponding Boolean Frame type. See Section 4.

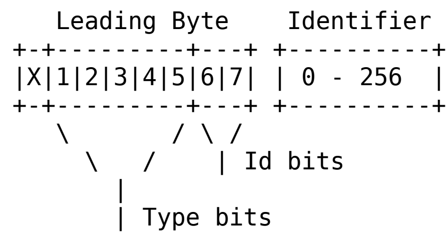

Figure 8: Boolean Frame

Identifier & Id bits: See Section 3 for details.

Type bits: See Section 4.

2.3.2. Integer Frames

Wide range of integer types are supported. Integer width and signedness are defined by choosing corresponding Integer Frame type. Signed integers are presented in two's complement notation.

Integer payload is always stored in big-endian format. See Section 1.1 for details.

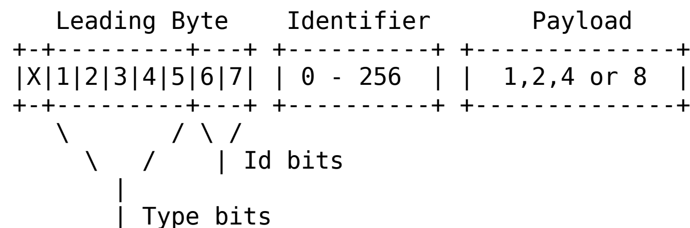

Figure 9: Integer Frames

Identifier & Id bits: See Section 3 for details.

Payload: Payload integer value.

Type bits: See Section 4.

2.3.3. Float Frames

Floating point number precision is defined by choosing corresponding Float Frame type. See Section 4 for frame types. Floats are presented in IEEE754 standard format and endianness is big-endian. See Section 1.1 for details.

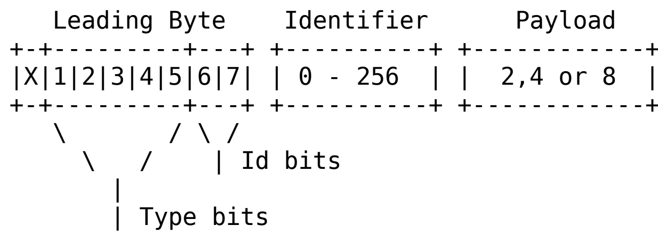

Figure 10: Float Frames

Identifier & Id bits: See Section 3 for details.

Payload: Payload float value.

Type bits: See Section 4.

2.3.4. String Frame

String Frame can hold UTF-8 encoded string. If implementation supports String Frame it must be able to validate UTF-8 encoding. See Section 1.2 for details.

Frame capacity is defined by selecting corresponding String Frame type. See Section 4 for details.

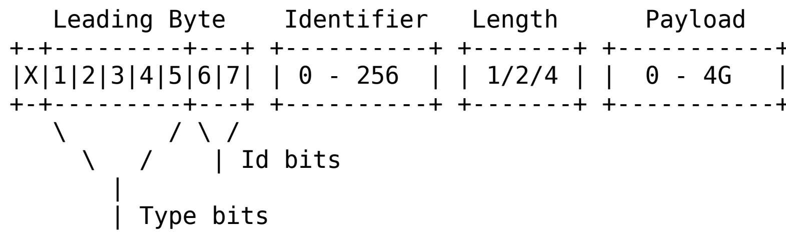

Figure 11: String Frame

Identifier & Id bits: See Section 3 for details.

Length: 8, 16, or 32-bits wide unsigned integer telling length of string in bytes. Depends on String Frame type, see Section 4 for details.

Payload: UTF-8 encoded string.

Type bits: See Section 4.

2.3.5. Binary Frame

Binary Frame holds arbitrary binary data.

Frame capacity is defined by selecting corresponding Binary Frame type. See Section 4 for details.

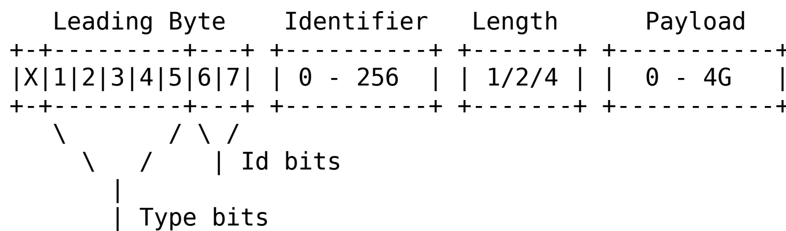

Figure 12: Binary Frame

Identifier & Id bits: See Section 3 for details.

Length: 8, 16, or 32-bits wide unsigned integer telling length of payload in bytes. Depends on Binary Frame type, see Section 4 for details.

Payload: Arbitrary binary data.

Type bits: See Section 4.

2.3.6. DateTime Frames

DateTime Frames hold date or date and time in UTC timescale as UTF-8 encoded string. String formats are compatible with RFC 3339 [RFC3339].

If implementation supports any of DateTime Frames it must be able to validate UTF-8 encoding. See Section 1.2 for details. Besides string formats must be validated but date data validation is not part of RSK. On writing phase illegal string format must be handled as error. On reading phase string format violation can be handled by rising warning and let user decide continue reading or not.

Date frame types and corresponding date string formats:

Date: YYYY-MM-DD

DateTime: YYYY-MM-DDTHH:MM:SSZ

DateTimeMillis: YYYY-MM-DDTHH:MM:SS.SSSZ

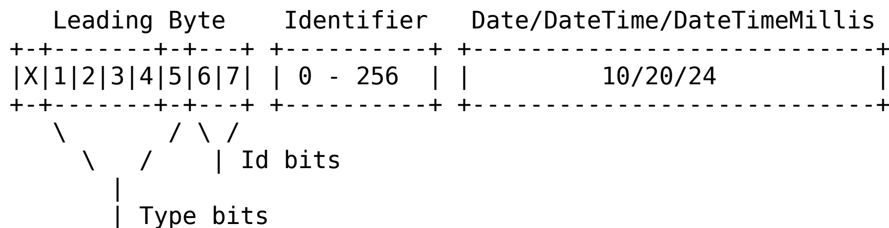

Figure 13: DateTime Frame

Identifier & Id bits: See Section 3 for details.

Date/DateTime/DateTimeMillis: Date, DateTime, or DateTimeMillis string depends of date frame type.

Type bits: See Section 4.

2.3.7. NTP Short Frame

NTP Short Frame holds NTP Short Format compatible timestamp. See RFC 5905 [RFC5905] for details.

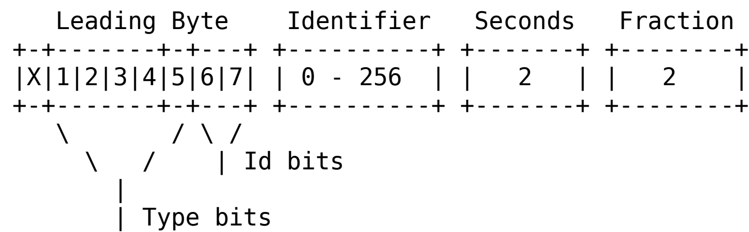

Figure 14: NTP Short Format Frame

Identifier & Id bits: See Section 3 for details.

Seconds: 16-bits unsigned integer telling seconds.

Fraction: 16-bits unsigned integer holding fractions of second.

Type bits: See Section 4.

2.3.8. NTP Timestamp Frame

NTP Timestamp Frame holds NTP Timestamp Format compatible timestamp. See RFC 5905 [RFC5905] for details.

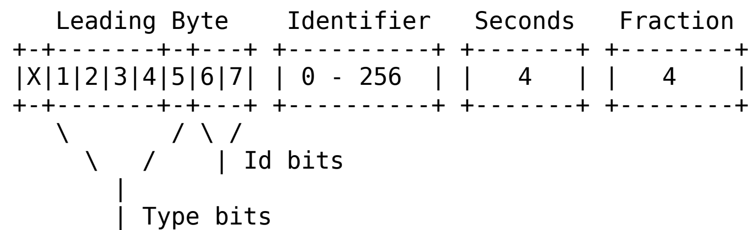

Figure 15: NTP Timestamp Format Frame

Identifier & Id bits: See Section 3 for details.

Seconds: 32-bits unsigned integer telling seconds.

Fraction: 32-bits unsigned integer holding fractions of second.

Type bits: See Section 4.

2.3.9. NTP Date Frame

NTP Date Frame holds NTP Date Format compatible date. See RFC 5905 [RFC5905] for details.

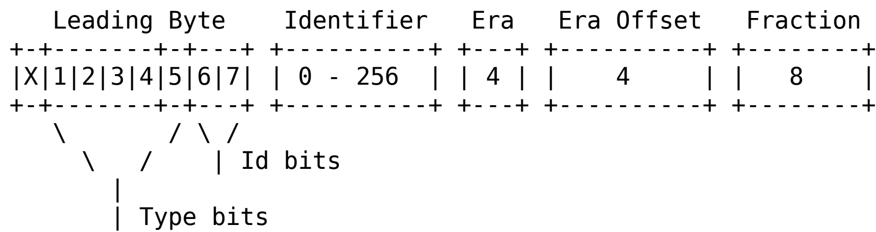

Figure 16: NTP Date Format Frame

Identifier & Id bits: See Section 3 for details.

Era: 32-bits signed integer telling the era of timestamp. See RFC 5905 [RFC5905] for era definitions.

Offset: 32-bits unsigned integer holding number of seconds since beginning of the Era.

Fraction: 64-bits unsigned integer holding fractions of second.

Type bits: See Section 4.

2.3.10. RSK Date Frame

RSK Date is optimized version of NTP Date Format defined in RFC 5905 [RFC5905].

Differences with NTP Date Format

Width of Era field: NTP Date Format has 32-bits wide Era field. Here Era is only 8-bits wide. Interpretation is same but with narrowed range. Epoch is same so Era 0 starts at 1900-01-01 00:00:00 UTC like in NTP Date Format.

Width of Fraction Field: NTP Date Format uses 64-bits wide Fraction field. Here fraction is only 16-bits wide and thus capable to 16 microsecond resolution.

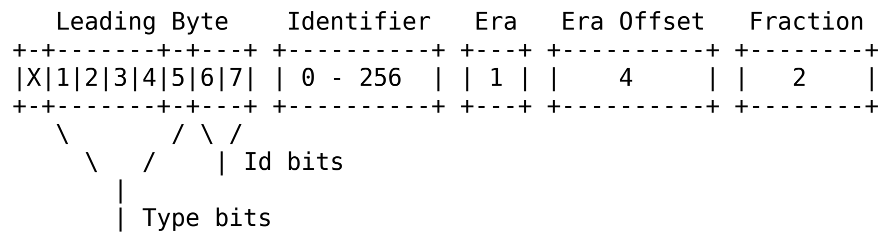

Figure 17: RSK Date Frame

Identifier & Id bits: See Section 3 for details.

Era: 8-bit signed integer telling the era of timestamp. See RFC 5905 [RFC5905] for era definitions.

Offset: 32-bit unsigned integer holding number of seconds since beginning of the Era.

Fraction: 16-bit unsigned integer holding fractions of second.

Type bits: See Section 4.

2.4. Extended Frames

Extended Frame is concept to introduce new structures and data types in future versions of RSK. The most significant bit in Leading Byte is reserved for Extended Frames. In this version Extended bit must not be set when writing a RSK document. If Extended Frame is discovered on reading phase it must be handles as error in this version.

3. Identifiers

All data frames and some of the meta frames can be tagged with an identifier. Identifier can be defined as 8 or 16-bit wide unsigned integer or as length-prefixed UTF-8 encoded string. If identifier is not needed it can be set to Null.

Frame's Leading Byte tells type of identifier. Identifier bytes are placed immediately after the Leading Byte. In case of integer identifier there is one or two bytes depending on selected integer identifier type. String identifier can take up to 256 bytes. See following sections for details.

3.1. Identifier Types in Leading Byte

Two least significant bits of Leading Byte are reserved for Id bits in all frame types which can be tagged with an identifier.

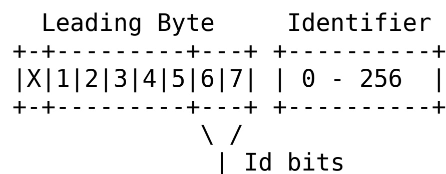

Figure 18: Identifier Types

Id bits values and identifier types:

00 Null Identifier. See Section 3.2.

01 8-bits wide Integer Identifier. See Section 3.3.

10 16-bits wide Integer Identifier. See Section 3.3.

11 String Identifier. See Section 3.4

3.2. Null Identifier

Some frames in a document may not need identifier so it can be left empty by setting it Null in Leading Byte.

3.3. Integer Identifiers

Integer identifier types are 8 or 16-bits wide unsigned integers. Integer identifiers are presented in big-endian format. See

Section 1.1 for details.

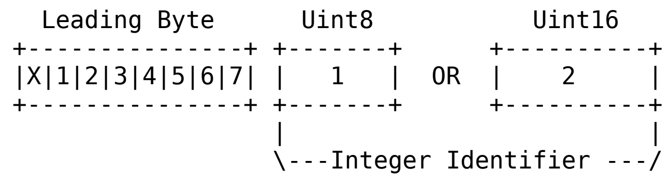

Figure 19: Integer Identifier

3.4. String Identifier

String identifier is length-prefixed and UTF-8 encoded. Length is presented by one byte as 8-bits wide unsigned integer at the beginning of identifier field. String identifier itself can be 0 - 255 bytes long.

If implementation supports string identifiers it must be able to validate UTF-8 encoding. See Section 1.2 for details.

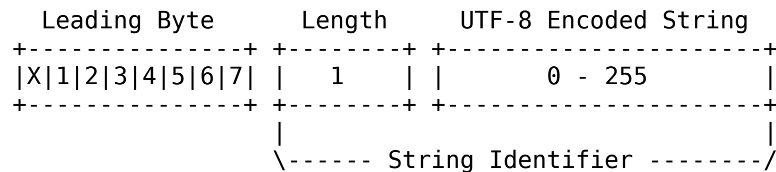

Figure 20: String Identifier

4. Frame Type Table

No	Frame	Type	Id	Array	Payload
1.	Null	0x00	[x]	[ ]	0
2.	Begin	0x04	[x]	[ ]	0
3.	End	0x08	[ ]	[ ]	0
4.	Boolean False	0x0C	[x]	[ ]	0
5.	Boolean True	0x10	[x]	[ ]	0
6.	TinyArray	0x14	[x]	[ ]	0 - 255 (items)
7.	Array	0x18	[x]	[ ]	0 - 64k (items)
8.	LongArray	0x1C	[x]	[ ]	0 - 4G (items)
9.	TinyString	0x20	[x]	[x]	0 - 255
10.	String	0x24	[x]	[x]	0 - 64k
11.	LongString	0x28	[x]	[x]	0 - 4G
12.	TinyBinary	0x2C	[x]	[x]	0 - 255
13.	Binary	0x30	[x]	[x]	0 - 64k
14.	LongBinary	0x34	[x]	[x]	0 - 4G
15.	Signed int 8-bits	0x38	[x]	[x]	1
16.	Signed int 16-bits	0x3C	[x]	[x]	2
17.	Signed int 32-bits	0x40	[x]	[x]	4
18.	Signed int 64-bits	0x44	[x]	[x]	8
19.	Unsigned int 8-bits	0x48	[x]	[x]	1
20.	Unsigned int 16-bits	0x4C	[x]	[x]	2
21.	Unsigned int 32-bits	0x50	[x]	[x]	4
22.	Unsigned int 64-bits	0x54	[x]	[x]	8
23.	Floating 16-bits	0x58	[x]	[x]	2
24.	Floating 32-bits	0x5C	[x]	[x]	4
25.	Floating 64-bits	0x60	[x]	[x]	8
26.	Date	0x64	[x]	[x]	10
27.	DateTime	0x68	[x]	[x]	20
28.	DateTimeMillis	0x6C	[x]	[x]	24
29.	NTP Short Format	0x70	[x]	[x]	4
30.	NTP Timestamp Format	0x74	[x]	[x]	8
31.	NTP Date Format	0x78	[x]	[x]	16
32.	RSK Date	0x7C	[x]	[x]	7

Frame Type Table columns:

Frame: Name of frame type. See Section 2 for detailed frame definitions.

Type: Hexadecimal value of Leading Byte with mask 0xFC. See Section 2.1 for detailed description of Leading Byte.

Id: All marked with X has identifier field. See Section 3 for details.

Array: All marked with X can be enclosed into a array. See Section 2.2.4 for details.

Payload: Payload length in bytes for data frames and item count for arrays.

5. Implementation Notes

RSK is designed so that implementations could have very small memory and other resource demands. Pay attention to memory usage and try to perform IO operations efficiently.

Implementations must make sure that well formed documents are written. On reading phase any deformation in document or frame structure must be detected and handled as error condition.

Implementations can vary depending on environment and usage. All implementations must support at least Begin and End Frames to be able to handle document structure. Other frame types may not be supported. Implementation may also support most or all frame types but not all identifier types. Some frame types can be also partially supported so that they can be detected and skipped on reading phase although their payload data is not interpreted.

6. Security Considerations

RSK is data encoding format and does not include any executable commands. Implementations must make sure that any parts of encoded documents are not leaked into execution memory. Even malformed documents must be handled so that memory leaks are avoided.

RSK does not include any means to validate payload data integrity. Protocols based on RSK or underlying mechanisms which are utilized by those protocols must take care of this. If data integrity is not checked can data get corrupted by malfunctioning devices, software, or malicious attackers.

7. IANA Considerations

The MIME media type for RSK documents is application/ruoska.

Type name: application

Subtype name: ruoska

Required parameters: n/a

Optional parameters: n/a

8. Normative References

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

Author's Address

Jukka-Pekka Makela
Janakkala, Tavastia Proper
Finland

Liite 2. Pakkausrajapinta

```
/**
 * \file ruoskaenc.h
 * \author Jukka-Pekka Mäkelä
 */

#ifndef RUOSKAENC_H
#define RUOSKAENC_H

#include "ruoska.h"

#ifdef __cplusplus
extern"C" {
#endif

/**
 * Ruoska Encoder struct is used with encoding methods to encode data.
 * Implementation is made private in this API.
 */
typedef struct ruoska_encoder RuoskaEncoder;

/**
 * Initializes Ruoska Encoder struct.
 *
 * \see Must be deleted by calling ruoska_delete_encoder.
 *
 * \param max_id_len Maximum string identifier length.
 * \param buffer_len Buffer size to be allocated for Encoder.
 * \param callback Pointer to callback function for actual writing.
 * \return Returns pointer to Encoder struct. Returns 0 on error.
 */
RuoskaEncoder* ruoska_init_encoder(
        uint8 max_id_len,
        RuoskaEncCallback callback,
        char* user_data);

RuoskaBool ruoska_reset_encoder(
        RuoskaEncoder* encoder);
/**
 * Deletes Ruoska Encoder struct and frees all allocated memory.
 *
 * \param encoder Pointer to Ruoska Encoder pointer to be deleted.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_delete_encoder(
        RuoskaEncoder** encoder);

/**
 * Gets active Encoder error code.
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns active error code.
 */
RuoskaErrorCode ruoska_enc_error(
        RuoskaEncoder* encoder);

/**
 * Sets identifier type to NULL.
 *
 * \see Identifier setters ruoska_set_id8(), ruoska_set_id16(),
```

```
 *      ruoska_set_str_id(), and ruoska_set_cstr_id().
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_set_null_id(
        RuoskaEncoder *encoder);

/**
 * Sets UINT8 identifier.
 *
 * \see Identifier setters ruoska_set_null_id(), ruoska_set_id16(),
 *      ruoska_set_str_id(), and ruoska_set_cstr_id().
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_set_id8(
        RuoskaEncoder *encoder,
        uint8 id);

/**
 * Sets UINT16 identifier.
 *
 * \see Identifier setters ruoska_set_null_id(), ruoska_set_id8(),
 *      ruoska_set_str_id(), and ruoska_set_cstr_id().
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_set_id16(
        RuoskaEncoder* encoder,
        uint16 id);

/**
 * Sets STRING identifier.
 *
 * \see Identifier setters ruoska_set_null_id(), ruoska_set_id8(),
 *      ruoska_set_id16(), and ruoska_set_cstr_id().
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_set_str_id(
        RuoskaEncoder* encoder,
        const char* str,
        uint8 str_len);

/**
 * Helper function to sets STRING identifier with c-string.
 *
 * \see Identifier setters ruoska_set_null_id(), ruoska_set_id8(),
 *      ruoska_set_id16(), and ruoska_set_str_id().
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_set_cstr_id(
        RuoskaEncoder* encoder,
        const char* str);
```

```
/**
 * Encodes BEGIN frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_begin(
        RuoskaEncoder* encoder);

/**
 * Encodes END frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_end(
        RuoskaEncoder* encoder);

/**
 * Encodes ARRAY frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param type Frame type for array items.
 * \param data
 * \param data_len Length of data in bytes.
 * \param capacity Capacity for array.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_array(
        RuoskaEncoder *encoder,
        RuoskaLeadingByte clb,
        ruoska_length item_count,
        RuoskaCapacity capacity);

/**
 * Encodes BOOLEAN frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Boolean value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_bool(
        RuoskaEncoder* encoder,
        RuoskaBool value);

/**
 * Encodes SIGNED INTERGER frame in dynamic fashion. Narrowest data type able
 * to store value is selected.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Signed integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_int(
        RuoskaEncoder* encoder,
        ruoska_int value);

/**
 * Encodes INT8 frame.
```

```
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Signed integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_int8(
        RuoskaEncoder* encoder,
        int8 value);

/**
 * Encodes INT16 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Signed integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_int16(
        RuoskaEncoder* encoder,
        int16 value);

#if RUOSKA_CONF_MAX_INT_WIDTH >= 32

/**
 * Encodes INT32 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Signed integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_int32(
        RuoskaEncoder* encoder,
        int32 value);

#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#if RUOSKA_CONF_MAX_INT_WIDTH == 64

/**
 * Encodes INT64 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Signed integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_int64(
        RuoskaEncoder* encoder,
        int64 value);

#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

/**
 * Encodes UNSIGNED INTERGER frame in dynamic fashion. Narrowest data type able
 * to store value is selected.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Unsigned integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
```

```
RuoskaBool ruoska_enc_uint(
        RuoskaEncoder* encoder,
        ruoska_uint value);
/**
 * Encodes UINT8 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Unsigned integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_uint8(
        RuoskaEncoder* encoder,
        uint8 value);

/**
 * Encodes UINT16 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Unsigned integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_uint16(
        RuoskaEncoder* encoder,
        uint16 value);

#if RUOSKA_CONF_MAX_INT_WIDTH >= 32

/**
 * Encodes UINT32 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Unsigned integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_uint32(
        RuoskaEncoder* encoder,
        uint32 value);
#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#if RUOSKA_CONF_MAX_INT_WIDTH == 64

/**
 * Encodes UINT64 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Unsigned integer value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_uint64(
        RuoskaEncoder* encoder,
        uint64 value);

#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#ifdef __STDC_IEC_559__

#if RUOSKA_CONF_MAX_FLOAT_WIDTH >= 32

/**
```

```
 * Encodes FLOAT32 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Float value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_float32(
        RuoskaEncoder* encoder,
        float value);

#endif /* RUOSKA_CONF_MAX_FLOAT_WIDTH */

#if RUOSKA_CONF_MAX_FLOAT_WIDTH == 64

/**
 * Encodes FLOAT64 frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param value Float value to be encoded.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_float64(
        RuoskaEncoder* encoder,
        double data);

#endif /* RUOSKA_CONF_MAX_FLOAT_WIDTH */

#endif /* __STDC_IEC_559__ */

/**
 * Encodes STRING frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param str String to be encoded.
 * \param str_len Length of string.
 * \param capacity
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_string(
        RuoskaEncoder* encoder,
        const char* str,
        ruoska_length str_len,
        RuoskaCapacity capacity);

/**
 * Encodes BINARY frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param data Binary data to be encoded.
 * \param data_len Length of binary data.
 * \param capacity
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_binary(
        RuoskaEncoder* encoder,
        const char* data,
        ruoska_length data_len,
        RuoskaCapacity capacity);
```

```
/**
 * Encodes DATE frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param date Date string.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_date(
        RuoskaEncoder* encoder,
        const char* date);

/**
 * Encodes DATETIME frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param date DateTime string.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_datetime(
        RuoskaEncoder* encoder,
        const char* datetime);

/**
 * Encodes DATETIME MILLIS frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param date DateTime with milliseconds string.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_datetime_millis(
        RuoskaEncoder* encoder,
        const char* datetime);

/**
 * Encodes NTP SHORT frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param seconds Seconds as unsigned integer.
 * \param fraction Fraction of second as unsigned integer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_ntp_short(
        RuoskaEncoder* encoder,
        uint16 seconds,
        uint16 fraction);

#if RUOSKA_CONF_MAX_INT_WIDTH >= 32

/**
 * Encodes NTP TIMESTAMP frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param seconds Seconds as unsigned integer.
 * \param fraction Fraction of second as unsigned integer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_ntp_timestamp(
        RuoskaEncoder* encoder,
        uint32 seconds,
```

```
        uint32 fraction);

#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#if RUOSKA_CONF_MAX_INT_WIDTH == 64

/**
 * Encodes NTP DATE frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param era Era as signed integer.
 * \param offset Era offset as unsigned integer.
 * \param fraction Fraction of second as unsigned integer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_ntp_date(
        RuoskaEncoder* encoder,
        int32 era,
        uint32 offset,
        uint64 fraction);

#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#if RUOSKA_CONF_MAX_INT_WIDTH >= 32

/**
 * Encodes RSK DATE frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \param era Era as signed integer.
 * \param offset Era offset as unsigned integer.
 * \param fraction Fraction of second as unsigned integer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_rsk_date(
        RuoskaEncoder* encoder,
        int8 era,
        uint32 offset,
        uint16 fraction);

#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

/**
 * Encodes NULL frame.
 *
 * \param encoder Ruoska Encoder pointer.
 * \return Returns true on success and false on failure.
 *         After failure get error code by ruoska_enc_error().
 */
RuoskaBool ruoska_enc_null(
        RuoskaEncoder* encoder);

#ifdef __cplusplus
}
#endif

#endif /* RUOSKAENC_H */
```

Liite 3. Purkurajapinta

```
/**
 * \file ruoskadec.h
 * \author Jukka-Pekka Mäkelä
 */
#ifndef RUOSKADEC_H
#define RUOSKADEC_H

#ifdef __cplusplus
extern"C" {
#endif

#include "ruoska.h"

/**
 * Ruoska Decoder struct is used with encoding methods to encode data.
 * Implementation is made private in this API.
 */
typedef struct ruoska_decoder RuoskaDecoder;

/**
 * Initializes Ruoska Decoder.
 *
 * \param id_buffer_size Maximum string identifier length.
 * \param read Pointer to callback function for reading input.
 * \param user_data Pointer to user specified data to idenfity decoder instance.
 * \return Returns pointer to Encoder struct. Returns 0 on error.
 */
RuoskaDecoder* ruoska_init_decoder(
        uint8 id_buffer_size,
        RuoskaDecCallback read,
        char* user_data);

/**
 * Deletes Ruoska Decoder.
 *
 * \param decoder Pointer to decoder pointer to be deleted.
 * \return Returns true if deletion succeeded otherwise false.
 */
RuoskaBool ruoska_delete_decoder(
        RuoskaDecoder** decoder);

/**
 * Gets active Decoder error code.
 *
 * \param decoder Pointer to decoder.
 * \return Returns active error code.
 */
RuoskaErrorCode ruoska_dec_error(
        RuoskaDecoder* decoder);

/**
 * Peeks type and identifier type of current frame.
 * Peeked frame is not consumed in the operation succeeded or not.
 *
 * \see Peek functions ruoska_peek_id(), ruoska_peek_str_id(),
 *      and ruoska_peek_size()
 *
 * \param decoder Pointer to decoder.
 * \param type Frame type pointer. Value is updated to current frame type.
 * \param id_type Identifier type pointer. Value is updated to identifier
```

```
 *                 type of current frame.
 * \return Returns true if peek operation succeeded otherwise false.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_peek(
        RuoskaDecoder* decoder,
        RuoskaFrameType* type,
        RuoskaIdType* id_type);

/**
 * Peeks type of current frame. Frame must have integer identifier.
 * Peeked frame is not consumed in the operation succeeded or not.
 *
 * \see Peek functions ruoska_peek(), ruoska_peek_str_id(),
 *      and ruoska_peek_size()
 *
 * \param decoder Pointer to decoder.
 * \param type Frame type pointer. Value is updated to current frame type.
 * \param id Integer identifier pointer. Value is updated by integer id.
 * \return Returns true if peek operation succeeded otherwise false.
 *         Operation fails if current frame does not have integer id.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_peek_id(
        RuoskaDecoder* decoder,
        RuoskaFrameType* type,
        uint16* id);

/**
 * Peeks type of current frame. Frame must have string indentifier.
 * Peeked frame is not consumed in the operation succeeded or not.
 *
 * \see Peek functions ruoska_peek(), ruoska_peek_id(), and ruoska_peek_size()
 *
 * \param decoder Pointer to decoder.
 * \param type Pointer to current frame type.
 * \param str Pointer to string id.
 * \param str_len Pointer to string length.
 * \return Returns true if peek operation succeeded otherwise false.
 *         Operation fails if current frame does not have string id.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_peek_str_id(
        RuoskaDecoder* decoder,
        RuoskaFrameType* type,
        char* str,
        uint8* str_len);

/**
 * Peeks size of current frame.
 * Peeked frame is not consumed in the operation succeeded or not.
 *
 * \see Peek functions ruoska_peek(), ruoska_peek_id(), and ruoska_peek_str_id()
 *
 * \param decoder Pointer to decoder.
 * \param type Pointer to current frame type.
 * \param str Pointer to string id.
 * \param str_len Pointer to string length.
 * \return Returns true if peek operation succeeded otherwise false.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_peek_size(
        RuoskaDecoder* decoder,
        ruoska_length* size);
```

```
/**
 * Skips current frame recursively.
 *
 * \param decoder Pointer to decoder.
 * \return Returns true skip operation succeeded otherwise false.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_skip(
        RuoskaDecoder* decoder);

/**
 * Returns current nesting level.
 *
 * \param decoder Pointer to decoder.
 * \return Returns current nesting level.
 */
uint8 ruoska_dec_nesting_level(
        RuoskaDecoder* decoder);

/**
 * Decodes BEGIN frame.
 *
 * \param decoder Pointer to decoder.
 * \return Returns true if current frame type is BEGIN and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_begin(
        RuoskaDecoder* decoder);

/**
 * Decodes END frame.
 *
 * \param decoder Pointer to decoder.
 * \return Returns true if current frame type is END and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_end(
        RuoskaDecoder* decoder);

/**
 * Decodes ARRAY frame.
 * Array items are decoded by type depending decoding functions.
 *
 * \param decoder Pointer to decoder.
 * \param type Pointer to frame type of items.
 * \param id_type Pointer identifier type of items.
 * \param item_count Pointer to item count.
 * \return Returns true if current frame type is ARRAY and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_array(
        RuoskaDecoder* decoder,
        RuoskaFrameType* type,
        RuoskaIdType* id_type,
        ruoska_length* item_count);

/**
 * Decodes SIGNED INTEGER frame.
 *
 * \param decoder Pointer to decoder.
 * \param data Pointer to integer payload.
```

```
 * \return Returns true if current frame type is SIGNED INTEGER and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_int(
        RuoskaDecoder* decoder,
        ruoska_int* data);

/**
 * Decodes UNSIGNED INTGER frame.
 *
 * \param decoder Pointer to decoder.
 * \param data Pointer to integer payload.
 * \return Returns true if current frame type is UNSIGNED INTEGER and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_uint(
        RuoskaDecoder* decoder,
        ruoska_uint* data);

#ifdef __STDC_IEC_559__

#if RUOSKA_CONF_MAX_FLOAT_WIDTH >= 32
/**
 * Decodes FLOAT32 frame.
 *
 * \param decoder Pointer to decoder.
 * \param data Pointer to float payload.
 * \return Returns true if current frame type is FLOAT32 and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_float(
        RuoskaDecoder* decoder,
        float* data);
#endif /* RUOSKA_CONF_MAX_FLOAT_WIDTH */

#if RUOSKA_CONF_MAX_FLOAT_WIDTH == 64
/**
 * Decodes FLOAT64 frame.
 *
 * \param decoder Pointer to decoder.
 * \param data Pointer to double payload.
 * \return Returns true if current frame type is FLOAT64 and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_double(
        RuoskaDecoder* decoder,
        double* data);
#endif /* RUOSKA_CONF_MAX_FLOAT_WIDTH */

#endif /* __STDC_IEC_559__ */

/**
 * Decodes STRING frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to string payload.
 * \param str Pointer to string length.
 * \return Returns true if current frame type is STRING and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
```

```
 */
RuoskaBool ruoska_dec_string(
        RuoskaDecoder* decoder,
        char* str,
        ruoska_length* str_len);

/**
 * Decodes BINARY frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to binary payload.
 * \param str Pointer to payload length.
 * \return Returns true if current frame type is BINARY and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_binary(
        RuoskaDecoder* decoder,
        char* data,
        ruoska_length* data_len);

/**
 * Decodes DATE frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to date payload.
 * \return Returns true if current frame type is DATE and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_date(
        RuoskaDecoder* decoder,
        char* date);

/**
 * Decodes DATETIME frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to datetime payload.
 * \return Returns true if current frame type is DATETIME and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_datetime(
        RuoskaDecoder* decoder,
        char* date);

/**
 * Decodes DATETIME MILLIS frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to datetime with milliseconds payload.
 * \return Returns true if current frame type is DATETIME MILLIS and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_datetime_millis(
        RuoskaDecoder* decoder,
        char* date);

/**
 * Decodes NTP SHORT frame.
 *
 * \param decoder Pointer to decoder.
```

```
 * \param str Pointer to seconds.
 * \param str Pointer to fraction of second.
 * \return Returns true if current frame type is NTP SHORT and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_ntp_short(
        RuoskaDecoder* decoder,
        uint16* seconds,
        uint16* fraction);

#if RUOSKA_CONF_MAX_INT_WIDTH >= 32
/**
 * Decodes NTP TIMESTAMP frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to seconds.
 * \param str Pointer to fraction of second.
 * \return Returns true if current frame type is NTP TIMESTAMP and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_ntp_timestamp(
        RuoskaDecoder* decoder,
        uint32* seconds,
        uint32* fraction);
#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#if RUOSKA_CONF_MAX_INT_WIDTH == 64
/**
 * Decodes NTP DATE frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to era.
 * \param str Pointer to era offset in seconds.
 * \param str Pointer to fraction of second.
 * \return Returns true if current frame type is NTP DATE and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_ntp_date(
        RuoskaDecoder* decoder,
        uint32* era,
        uint32* offset,
        uint64* fraction);
#endif /* RUOSKA_CONF_MAX_INT_WIDTH */

#if RUOSKA_CONF_MAX_INT_WIDTH >= 32
/**
 * Decodes RSK DATE frame.
 *
 * \param decoder Pointer to decoder.
 * \param str Pointer to era.
 * \param str Pointer to era offset in seconds.
 * \param str Pointer to fraction of second.
 * \return Returns true if current frame type is RSK DATE and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_rsk_date(
        RuoskaDecoder* decoder,
        int8* era,
        uint32* offset,
        uint16* fraction);
#endif /* RUOSKA_CONF_MAX_INT_WIDTH */
```

```
/**
 * Decodes NULL frame.
 *
 * \param decoder Pointer to decoder.
 * \return Returns true if current frame type is NULL and it is
 *         successfully decoded.
 *         After failure get error code by ruoska_dec_error().
 */
RuoskaBool ruoska_dec_null(
        RuoskaDecoder* decoder);

#ifdef __cplusplus
}
#endif

#endif /* RUOSKADEC_H */
```

Liite 4. Yksikkötesti esimerkkejä

TEST makro luo test_enc_int_ids() funktion, jota kutsumalla testi voidaan suorittaa. Makro luo myös käyttäjälle näkymättömiä funktioita, joiden avulla tuotetaan testin tulosnäkymä.

```
TEST(test_enc_int_ids)
{
    /* Identifiers */
    uint8 id8 = 0x1D;
    uint16 id16 = 0xFACE;

    static const byte document[] =
    {
        0x04 | 0x01, /* RUOSKA_FRAME_BEGIN with UINT8 Identifier */
        0x1D,        /* UINT8 Identifier */
        0x04 | 0x02, /* RUOSKA_FRAME_BEGIN with UINT16 Identifier */
        0xFA, 0xCE,  /* UINT16 Identifier */
        0x08,        /* RUOSKA_FRAME_END  */
        0x08         /* RUOSKA_FRAME_END  */
    };

    /* Set uint8 id. */
    ruoska_set_id8(encoder, id8);
    CHECK(encoder->id_type == RUOSKA_ID_UINT8);
    CHECK((uint8)encoder->id->data[0] == id8);
    CHECK(encoder->id->data_len == RUOSKA_INT8_WIDTH);

    /* Encode rooting Begin. Identifier settings should not be changed. */
    ruoska_enc_begin(encoder);
    CHECK(encoder->id_type == RUOSKA_ID_UINT8);
    CHECK((uint8)encoder->id->data[0] == id8);
    CHECK(encoder->id->data_len == RUOSKA_INT8_WIDTH);

    /* Set uint16 id. Start sub branch. */
    ruoska_set_id16(encoder, id16);
    ruoska_enc_begin(encoder);
    CHECK(encoder->id_type == RUOSKA_ID_UINT16);
    CHECK((uint8)encoder->id->data[0] == 0xFA);
    CHECK((uint8)encoder->id->data[1] == 0xCE);
    CHECK(encoder->id->data_len == RUOSKA_INT16_WIDTH);

    /* End sub branch and document. */
    ruoska_enc_end(encoder);
    ruoska_enc_end(encoder);

    print_array(output, 7);
    print_array(document, 7);

    CHECK(memcmp(output, document, 7) == 0);
}
```

Testien tuloksen voivat näyttää seuraavalta:

```
Ruoska Encoder Tests!
------------------------------------------------
Test [ test_encoder_init_and_delete ]
Result: PASS
------------------------------------------------
Test [ test_enc_smallest_document ]
Result: PASS
------------------------------------------------
Test [ test_enc_int_ids ]
[ 05 1D 06 FA CE 08 08 ]
[ 05 1D 06 FA CE 08 08 ]
Result: PASS
------------------------------------------------
Test [ test_enc_str_id ]
[ 07 10 48 61 70 70 79 20 49 64 65 6E 74 69 66 69 65 72 08 ]
[ 07 10 48 61 70 70 79 20 49 64 65 6E 74 69 66 69 65 72 08 ]
Result: PASS
------------------------------------------------
```